8-2005

Studies in Rheology: Molecular Simulation and Theory

Chunggi Baig
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a dissertation written by Chunggi Baig entitled "Studies in Rheology: Molecular Simulation and Theory." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Chemical Engineering.

Hank D. Cochran, Major Professor

We have read this dissertation and recommend its acceptance:

Brian J. Edwards, David J. Keffer, Robert J. Hinde

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a dissertation written by Yihua Bai entitled "High Performance Parallel Approximate Eigensolver for Real Symmetric Matrices." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Robert C. Ward

Major Professor

We have read this dissertation
and recommend its acceptance:

Michael W. Berry

Jack J. Dongarra

Robert J. Hinde

Accepted for the Council:

Anne Mayhew
Vice Chancellor and
Dean of Graduate Studies

(Original signatures are on file with official student records.)

High Performance
Parallel Approximate Eigensolver
for Real Symmetric Matrices

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Yihua Bai

December 2005

Dedication

This dissertation is dedicated to my wonderful husband Guoping for his love and support.

Acknowledgement

I would like to thank my advisor, Dr. Robert C. Ward, for his guidance, motivation and support throughout my PhD study at the University of Tennessee. I thank him for giving me the opportunity to work in his group. My experience there is really enjoyable.

I also wish to thank Dr. Michael Berry, Dr. Jack Dongarra from Computer Science Department and Dr. Robert J. Hinde from Chemistry Department for serving on my PhD committee and giving valuable suggestions on my dissertation.

In addition, I would like to thank Dr. Wilfried Gansterer, now assistant professor at the University of Vienna, for providing a lot of technical details on the sequential block tridiagonal divide-and-conquer algorithm and code. Many helpful and instructive discussions and communications are truly appreciated.

Finally I wish to express my appreciation to Dr. Richard P. Muller of Sandia National Laboratories and Dr. Guoping Zhang of Indiana State University for providing application test matrices in this dissertation.

Abstract

In the first-principles calculation of electronic structures, one of the most time-consuming tasks is that of computing the eigensystem of a large symmetric nonlinear eigenvalue problem. The standard approach is to use an iterative scheme involving the solution to a large symmetric linear eigenvalue problem in each iteration. In the early and intermediate iterations, significant gains in efficiency may result from solving the eigensystem to reduced accuracy. As the iteration nears convergence, the eigensystem can be computed to the required accuracy.

Traditional real symmetric eigensolvers compute the eigensystem in three steps: 1) reduce a dense matrix to a symmetric tridiagonal form using orthogonal transformations; 2) compute eigenpairs of the tridiagonal matrix; 3) back-transform eigenvectors of the tridiagonal matrix to those of the original matrix. Stable and efficient eigen-decomposition algorithms for symmetric tridiagonal matrix are under constant investigation, while the performance of orthogonal reduction step remains a bottleneck.

The main contribution of this dissertation is an efficient parallel approximate eigensolver that computes eigenpairs of a real symmetric matrix to reduced accuracy. This eigensolver consists of three major parts: 1) a parallel block tridiagonal divide-and-conquer algorithm that computes the approximate eigenpairs of a block tridiagonal matrix to prescribed accuracy; 2) a parallel block tridiagonalization algorithm that constructs a block tridiagonal matrix from a sparse matrix or "effectively" sparse matrix – matrix with many small elements that can be regarded as zeros without affecting the prescribed accuracy of the eigenvalues; 3) a parallel orthogonal block tridiagonal reduction algorithm that reduces a dense real symmetric matrix to block tridiagonal form using similarity transformations with a high ratio of level 3 BLAS operations. The parallel approximate eigensolver chooses a proper combination of the three algorithms depending on the structure of the input matrix and computes all the eigenpairs of the input matrix to prescribed accuracy.

Numerical results show that the parallel block tridiagonal divide-and-conquer algorithm is very efficient when at least a few off-diagonal blocks have a relatively low

rank. With a very low computational cost, the parallel block tridiagonalization algorithm constructs a block tridiagonal matrix from a sparse or "effectively" sparse input matrix. The parallel orthogonal block tridiagonal reduction algorithm achieves high performance due to high ratio of level 3 BLAS operations. Using a small block size for the parallel orthogonal block tridiagonal reduction algorithm is a critical factor for competitive performance when combined with the parallel block tridiagonal divide-and-conquer algorithm.

Our parallel approximate eigensolver has the limitation that the block tridiagonal matrices, either as the input matrices or after pre-processing steps, should have off-diagonal blocks with low rank, say 20 or less, or a very high ratio of deflation to achieve satisfactory performance. In addition, large variation in deflation rate may lead to workload imbalance, although such cases appear to be rare. Future work may include a complete data parallel implementation of the block tridiagonal divide-and-conquer algorithm and a parallel adaptive eigensolver that detects matrix structure automatically, adjusts the accuracy requirement when necessary and chooses the proper algorithms to solve the eigenproblem.

Table of Contents

List of Tables

List of Figures

List of Notation

A	Real symmetric matrix of order n
A_{ij}	(i, j)-th submatrix of A
B_i	The *i*-th diagonal block of a block tridiagonal matrix, $1 \le i \le q$
C_i	The *i*-th off-diagonal block of a block tridiagonal matrix, $1 \le i \le q-1$
D	Real symmetric block diagonal matrix of order n
D_i	The *i*-th diagonal block of D, $1 \le i \le q$
E	Error matrix of order n
$\mathcal{G}_i$	The *i*-th subgrid assigned to the *i*-th diagonal block of a block tridiagonal matrix, $1 \le i \le q$
H	Householder transformation matrix
I	Identity matrix of order n
M	Real symmetric block tridiagonal matrix of order n
P	Permutation matrix of order n
Q	Orthogonal transformation matrix
T	Real symmetric tridiagonal matrix of order n
V	Eigenvector matrix of M
X	Eigenvector matrix of A
Λ	Diagonal matrix of eigenvalues
Σ	Diagonal matrix of singular values
a_{ij}	(i, j)-th element of matrix A
b_i	Size of a diagonal block B_i, $1 \le i \le q$
c	Number of processor columns in a 2D processor grid
e_i	Vector with all zeros except that the *i*-th element is 1.
m	Matrix rows . For real symmetric matrices, $m = n$
n	Matrix columns

List of Notation (continued)

n_b	Row and column block size for parallel 2D block cyclic distribution of square matrices
p	Number of processor
p_b	Panel width for column block operation
p_i	Number of processors in processor grid $\mathcal{G}_i$, $1 \le i \le q$
q	Number of diagonal blocks in real symmetric block tridiagonal matrix M
r	Number of processor rows in a 2D processor grid
t	Time complexity
α	Start up time of a data transfer
β	Time to transfer a double precision floating-point number
ε_{mach}	Machine precision
δ_{ij}	(i, j)-th element of identity matrix, $\delta_{ii} = 1$ and $\delta_{ij} = 0, \forall i \ne j$
γ	Time for one floating-point computation
λ_i	i-th eigenvalue of a matrix, $1 \le i \le n$
ρ_i	Approximate rank of C_i, $0 \le \rho_i \le \min(b_i, b_{i+1})$
σ_i	i-th singular value of a matrix
τ	Tolerance parameter, $\varepsilon_{mach} \le \tau < 0.1$

1 Introduction and background

To construct an efficient and flexible eigensolver for real symmetric matrices is a challenging task because users with different backgrounds in the scientific community have distinctive requirements.

Practical applications generate real symmetric matrices of different kinds. For example, dense versus sparse and structured versus non-structured. Requirements for the matrix eigen-decomposition are also different. Some applications require only the eigenvalues, and some require the full set of eigenvalues and eigenvectors, while still others require only a few selected eigenpairs. In addition to the requirements from the applications, current hardware capabilities may also limit how many eigenpairs are computed and what eigen-decomposition algorithms are used.

In the first-principles calculation of electronic structures, the Schrödinger equation

$$\mathcal{H}\Phi = \mathcal{E}\Phi \tag{1.1}$$

is solved approximately. Here $\mathcal{H}$ is a Hermitian operator called the Hamiltonian, Φ is the wave function of electrons, and $\mathcal{E}$ is the electronic energy. This equation is intrinsically an eigenvalue problem because both Φ and $\mathcal{E}$ are unknown. The Schrödinger equation contains all the necessary information of physical systems of particles. These systems may have many electrons and nuclei whose interactions are often coupled. The study of such a complex system is called many-body problem. Except for some very simple systems like hydrogen atom, there is no way to get an exact solution for them.

One of the widely used approximation methods for solving Equation 1.1 is called the Hartree-Fock method. In this method, the many-electron system is approximated by an effective one-electron system, where all other electrons are considered as effective background. The many-body problem is thus reduced to a single-body problem [71]. The resultant Hartree-Fock equation is a non-linear integro-differential equation containing the desired unknown energy levels and wave functions. This equation is further converted into a non-linear symmetric eigenvalue problem and solved by an iterative procedure

called the self-consistent field (SCF) method (see Section 1.2 for details). In each iteration of the SCF procedure, a linear real symmetric eigenvalue problem is solved. In early and intermediate iterations, it may be more efficient to compute the eigenpairs to reduced accuracy [91]. As the SCF iterations near convergence, eigenpairs are computed to the required accuracy.

As the size of the system to be modeled and the requirements for the resolution of answers increase, the magnitude of the computational problem increases significantly. Solutions can soon only be obtained through the use of parallel and distributed computation, which in turn requires either parallelization of sequential algorithms or design of new parallel algorithms.

The goal of this dissertation is to develop an efficient parallel approximate eigensolver for real symmetric matrices that chooses appropriate algorithms according to different matrix structures and user-specified parameters such as accuracy tolerance.

1.1 Problem statement

For a real symmetric matrix $A \in \mathbb{R}^{n \times n}$ and an accuracy tolerance τ, we design and implement an efficient parallel approximate eigensolver that computes the approximate eigenpairs of A to the prescribed accuracy tolerance τ bounded by $\varepsilon_{mach} \le \tau < 0.1$. That is, we compute X and Λ such that

$$A \approx X \Lambda X^T$$

where X contains the approximate eigenvectors, the diagonal matrix Λ contains the approximate eigenvalues, X and Λ satisfy

$$\left\| A - X \Lambda X^T \right\|_2 = O\left(\tau \left\| A \right\|_2 \right),$$

and X is numerically orthogonal, i.e.,

$$\max \left\| \left(X X^T - I \right) e_i \right\|_2 = O\left(\varepsilon_{mach} n \right), \quad 1 \le i \le n.$$

When high accuracy is required, an existing reliable eigensolver like those found in ScaLAPACK [13] will be used; when lower accuracy suffices, then other algorithms

based upon the block tridiagonal eigensolver [43] may be more efficient. Thus, a major task for our parallel approximate eigensolver is to construct a symmetric block tridiagonal matrix

$$M = \begin{pmatrix} B_1 & C_1^T & & & \\ C_1 & B_2 & C_2^T & & \\ & C_2 & B_3 & \ddots & \\ & & \ddots & \ddots & C_{q-1}^T \\ & & & C_{q-1} & B_q \end{pmatrix},$$

which is an approximation to A, and to compute approximate eigenpairs of M efficiently. The construction of M is implemented either by orthogonal transformations or by alternative methods, depending on properties of A and the required accuracy.

1.2 Application and motivation

In quantum chemistry, material science and physics, electronic properties determine the structure-property relationship of a specific material and are fully contained in the electronic wave functions. The wave function of an electron in a molecule is called the molecular orbital. These wave functions are fundamentally difficult to obtain. Different approximation methods have been developed to compute electronic wave functions by solving the Schrödinger equation (Equation 1.1) approximately, e.g., the Hartree-Fock method [91, 17], density functional method [62, 78], and perturbation method [70]. Each of those methods is appropriate for a specific application area. An important one of those methods is the Hartree-Fock self-consistent method, which is used for electronic structure calculations in quantum chemistry, condensed matter physics, optics, etc. Since the Hartree-Fock equation is a non-linear differential equation, a molecular orbital is expanded in terms of a linear combination of a set of basis functions, so that the Hartree-Fock equation can be represented in matrix form. The resultant equation is called the Roothaan equation [91],

$$F(C)C = SC\mathcal{E}, \qquad (1.2)$$

where $F(C)$, C, S and $\mathcal{E}$ are the Fock matrix, the eigenvector matrix, the overlap matrix between basis functions, and the diagonal matrix of eigenvalues, respectively. The eigenvector matrix contains the coefficients for the wave functions, and the eigenvalues are electronic energies. The matrix S is positive definite.

To compute the coefficients for the wave functions that best describe molecular orbitals, one needs to solve the Roothaan equation, typically by the self-consistent field (SCF) method as shown in Figure 1.1. In the SCF procedure, Equation 1.2 is first reduced to a standard non-linear real symmetric eigenvalue problem

$$F'C' = C'\mathcal{E}, \tag{1.3}$$

where $F' = U^{-1}F(C)U^{-T}$, $C' = U^T C$ and U comes from a factorization of the overlap matrix $S = UU^T$. Then Equation 1.3 can be solved iteratively until convergence (or self-consistency) is achieved. In each iteration, after C' is computed, a new F' is computed as a function of C'; thus, a new Equation 1.3 is solved.

One criterion that can be used for convergence is the total electronic energy of each iteration, i.e., the difference between the total electronic energies of two successive iterations should be bounded by a prescribed tolerance. For a system with N electrons,

SCF Procedure

1) *Initial guess of wave functions* C
2) *Factorize overlap matrix* $S = UU^T$
3) *do*
   3.1) *Normalize Fock matrix*: $F' = U^{-1}FU^{-T}$
   3.2) *Compute* $C' = U^T C$
   3.3) *Solve* $F'C' = C'\mathcal{E}$
   3.4) *Compute new* $C = U^{-T}C'$
   3.5) *If not converge, construct new* F, *goto* 3.1.

Figure 1.1 Self-consistent field (SCF) procedure.

this total electronic energy equals $2\sum_{i=1}^{N/2}\mathcal{E}_{i,i}$ if N is even, and $2\sum_{i=1}^{(N-1)/2}\mathcal{E}_{i,i}+\mathcal{E}_{(N+1)/2,(N+1)/2}$ if N is odd. Theoretically, in order to guarantee an exact solution of wave functions, the number of basis functions must be infinite. Practically, only a finite number of basis functions can be used. The size of the matrix generated is determined by the number of bases. For N electrons in a molecule, at least N basis functions are needed to represent the molecule. As the number of molecules and electrons to be modeled increases, the number of bases becomes larger, and so does the corresponding matrix size. Thus, a □□荊 鍥□ 脈↑□ 愁 %n the SCF method is to solve large symmetric eigenvalue problems efficiently in each iteration.

We will use the SCF method in electronic structure calculations as our model problem. The sizes of our test matrices from quantum physics and chemistry range from moderate to large. Also, random matrices and matrices with specific eigenvalue distributions will be generated for testing specific properties of the eigensolver and very large problems.

1.3 Brief review of related work

Real symmetric eigenvalue problems have been studied intensively and extensively [26, 49, 77, 96]. Different algorithms have been developed for solving effectively and efficiently problems with different properties and requirements, such as dense matrices, sparse matrices, full spectrum required, or partial eigensystem required.

As the processors manufactured today become more powerful, the gap between CPU speed and memory access time has become much greater. To minimize this effect, algorithms have been reconstructed to take advantage of the deep memory hierarchy of modern computers and distributed data storage in parallel computers. For example, numerical software packages like LAPACK [1] and ScaLAPACK [13] implement linear algebra software using blocked algorithms to increase the number of floating-point operations per data access by maximizing the use of level 3 BLAS operations. References are made to such algorithms below as current algorithms are briefly described.

Traditional real symmetric eigensolvers for dense matrices decompose a real

symmetric matrix in three steps:

1) Reduce a dense matrix into a symmetric tridiagonal form using orthogonal transformations.

LAPACK currently implements the reduction in one step [35]. First, a sequence of k Householder transformations is computed and accumulated, which involves matrix-vector multiplications, that is, level 2 BLAS operations [34]. Then the rest of the matrix A is updated using a symmetric rank-2k update, which is a level 3 BLAS operation [33]. The level 2 BLAS operations count for about 50% of the total floating-point operations in the reduction to tridiagonal form.

Successive Bandwidth Reduction (SBR) [8, 12, 11] implements the reduction in two steps. First, a dense matrix is reduced to a banded form using mostly level 3 BLAS operations, and then the banded matrix is reduced to tridiagonal form using mostly level 2 BLAS operations. This approach has a more favorable data access pattern and a higher ratio of level 3 BLAS operations. However, the total amount of floating-point operations of SBR is higher than that of the LAPACK reduction algorithm. In addition, when the eigenvectors are required, the back transformation from SBR results in more storage space and higher computational complexity.

2) Compute eigenpairs of the tridiagonal matrix.

Let $T \in \mathbb{R}^{n \times n}$ be a real symmetric tridiagonal matrix; some of the frequently used algorithms for computing its eigensystem are described below:

2.1) Symmetric QR iteration with shift [40] as shown in Figure 1.2 is a stable method and still commonly used to compute all eigenpairs of T. The computational complexity of the symmetric QR algorithm for computing all eigenvalues and eigenvectors is $O\left(n^3\right)$.

The shift in the QR iteration is used to speed up the convergence [97]. The algorithm is typically implemented in an implicit form using a double shift, avoiding the potential numerical error and complex arithmetic in the above

$T_0 = T$
for $k = 1, 2, \ldots$
choose shift μ_k
compute QR factorization
$Q_k R_k = T_{k-1} - \mu_k I$
$T_k = R_k Q_k + \mu_k I$
end

Figure 1.2 QR iteration.

formulation [84].

The QR algorithm is sequential in nature. Parallel implementations of the QR algorithm have been developed in an attempt to exploit more parallelism [3, 61, 63, 93, 69], for example, by adjusting the sequential algorithm [93] or by pipelining the computation [63].

2.2) The Divide-and-conquer algorithm [16, 24, 48, 87, 92] is typically more than twice as fast as the symmetric QR [92] and also computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix. The computational complexity of the divide-and-conquer algorithm is also $O\left(n^3\right)$ in the worst case.

The matrix order of the problem is reduced by re-writing T as

$$T = \begin{bmatrix} T_1 & O \\ O & T_2 \end{bmatrix} + \delta u u^T ,$$

where $T_1 \in n_1 \times n_1$ and $T_2 \in n_2 \times n_2$ are tridiagonal matrices, $n_1 + n_2 = n$, $\delta = t_{n_1+1, n_1}$ and $u = \left(\underbrace{0, \cdots 0, 1}_{n_1}, \underbrace{1, 0, \cdots, 0}_{n_2} \right)$. We then compute the eigen-decomposition of the smaller subproblems T_1 and T_2 to obtain $T_1 = Q_1 \Lambda_1 Q_1^T$ and $T_2 = Q_2 \Lambda_2 Q_2^T$. Now, we have

$$T = \begin{bmatrix} Q_1 \Lambda_1 Q_1^T & O \\ O & Q_2 \Lambda_2 Q_2^T \end{bmatrix} + \delta u u^T$$

$$= \begin{bmatrix} Q_1 & O \\ O & Q_2 \end{bmatrix} \left(D + \delta y y^T \right) \begin{bmatrix} Q_1^T & O \\ O & Q_2^T \end{bmatrix},$$

where $D = \begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix}$ and $y = \begin{bmatrix} Q_1^T & O \\ O & Q_2^T \end{bmatrix} u$. To decompose $D + \delta y y^T$ into $V \Lambda V^T$, the eigenvalues Λ of $D + \delta y y^T$ can be computed by solving the secular equation $f(\lambda) = 1 + \delta \sum_{j=1}^{n} \frac{y_j^2}{d_j - \lambda} = 0$ efficiently and stably [66, 72].

For a computed eigenvalue $\hat{\lambda}_i$, its corresponding eigenvector $\hat{v}_i$ can be computed by

$$\hat{v}_i = \frac{\left(D - \hat{\lambda}_i I \right)^{-1} y}{\left\| \left(D - \hat{\lambda}_i \right)^{-1} y \right\|_2}. \tag{1.4}$$

However, with close eigenvalues, eigenvectors computed with this formula will lose their orthogonality [89, 24]. Fortunately, there is a numerically stable method to compute the orthogonal eigenvectors without using extended precision [52, 51]. First, the computed eigenvalues $\hat{\lambda}_i$ are taken as the exact eigenvalues of another matrix $D + \delta \overline{y} \overline{y}^T$. Each component of $\overline{y}$ can be computed by

$$\overline{y}_i = \sqrt{\left(\hat{\lambda}_i - d_i \right) \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{\hat{\lambda}_j - d_i}{d_j - d_i}}. \tag{1.5}$$

Then vector y in Equation 1.4 is replaced by $\overline{y}$ in Equation 1.5, and the

eigenvectors of $D+\delta yy^T$ are computed by

$$\hat{v}_i = \frac{\left(D-\hat{\lambda}_i I\right)^{-1}\bar{y}}{\left\|\left(D-\hat{\lambda}_i\right)^{-1}\bar{y}\right\|_2} = \frac{\left(\frac{\bar{y}_1}{d_1-\hat{\lambda}_i},\cdots,\frac{\bar{y}_n}{d_n-\hat{\lambda}_i}\right)^T}{\sqrt{\sum_{j=1}^{n}\frac{\bar{y}_j^2}{\left(d_j-\hat{\lambda}_i\right)^2}}}.$$

The eigenvector matrix of T can then be computed by $Q=\begin{pmatrix} Q_1 & \\ & Q_2 \end{pmatrix}V$, with the eigenvalues on the appropriate diagonals of Λ.

In this recursive algorithm as shown in Figure 1.3, the matrix T is divided into submatrices recursively until the submatrices are small enough to be solved quickly using other stable methods.

The divide-and-conquer algorithm is considered inherently parallel. However, its parallel implementation is a challenging task [36, 44, 58, 92]. One needs to handle deflation (see Section 2.1.3) properly to minimize floating-point operation count and communication cost and maintain workload balance, all at the same time [92].

2.3) Bisection and Inverse iteration [49] is able to compute selected eigenpairs of T. The worst-case computational complexity is $O\left(n^3\right)$ when all eigenpairs are computed.

Define the polynomial $p_r(\mu)=\det(T_r-\mu I)$ [49], where $r=1,2,\cdots,n$ and $T_r=T_{(1:r,1:r)}$. Set $p_0(\mu)=1$, $p_1(\mu)=t_{1,1}-\mu$. For $r=2,3,\cdots,n$, $p_r(\mu)$ can be expressed recursively as $p_r(\mu)=(t_{r,r}-\mu)p_{r-1}(\mu)-t_{r,r-1}^2 p_{r-2}(\mu)$.

subroutine $[Q,\Lambda] = D\&C(T)$

if T *is small*

$\quad$ *solve* $T = V\Lambda V^T$

$\quad$ *return*

else

$\quad T = \begin{bmatrix} T_1 & O \\ O & T_2 \end{bmatrix} + \delta uu^T$

$\quad [Q_1,\Lambda_1] = D\&C(T_1)$

$\quad [Q_2,\Lambda_2] = D\&C(T_2)$

$\quad$ *construct* $D + \delta yy^T,\ D = \begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix}$

$\quad$ *decompose* $D + \delta yy^T = V\Lambda V^T$

$\quad$ *compute* $Q = \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} V$

$\quad$ *return* $[Q,\Lambda]$

end

Figure 1.3 Divide-and-conquer (D&C) algorithm.

The sequence $\{p_0(\mu), p_1(\mu), \cdots, p_n(\mu)\}$ forms a Sturm sequence of polynomials; the root of $p_n(\mu)$ can be found in $O(n^2)$ time complexity using the bisection method [47].

After an eigenvalue λ has been computed by the bisection method, the corresponding eigenvector can be computed by inverse iteration [80, 57, 59] as shown in Figure 1.4.

Reorthogonalization is required to compute orthogonal eigenvectors when the eigenvalues form a tight cluster, i.e., the gap between any two eigenvalues in the cluster is small [77]. That may lead to $O(n^3)$ computational complexity in the worst case.

$$
\begin{aligned}
&v^{(0)} = b \\
&for \quad k = 1, 2, \ldots \\
&\quad solve \;\; (T - \lambda I) v^{(k+1)} = v^{(k)} \\
&\quad v^{(k+1)} = v^{(k+1)} / \left\| v^{(k+1)} \right\| \\
&end
\end{aligned}
$$

Figure 1.4 Inverse iteration.

In parallel implementation, sometimes a multisection algorithm is used to compute the eigenvalues [6, 69]. When eigenvalues are well separated, then the eigenvectors can be computed independently without communication. For clustered eigenvalues, reorthogonalization may be necessary and involves significant communication if those eigenvectors are not on the same processor.

2.4) The Multiple Relatively Robust Representations (MRRR) algorithm [29, 30, 32] typically computes all the eigenvectors in $O\left(n^2\right)$ time without explicit reorthogonalization.

First a relatively robust representation of T is computed in the form of $T + \mu I = LDL^T$ where $T + \mu I$ is positive definite. Based on the fact that eigenvalues are less sensitive to perturbations in off-diagonal entries of a bidiagonal matrix [76, 39], the eigenvalues of T can be computed to high relative accuracy using this representation. For clustered eigenvalues, a new shift that is close to the clustered eigenvalues is used to compute a new relatively robust representation [31, 75]. A twisted factorization [29, 31] is computed to find which equation of the near singular system $\left(T - \hat{\lambda} I\right)\hat{v} = 0$ is to be neglected so that an accurate eigenvector can be calculated. Finally, differential variants of the quotient-difference algorithm [39, 85, 86] is used to compute both the accurate eigenvalues and numerically stable twisted factorizations. Figure 1.5 shows the important steps in the MRRR algorithm.

Choose shift μ *for an RRR*
Compute RRR: $T + \mu I = L_p D_p L_p^T$
**Compute eigenvalues from* $L_p D_p L_p^T$ *to high accuracy*
if eigenvalues are isolated
  for each isolated eigenvalues
    compute twisted factorization
    find the equation to neglect
    compute eigenvectors
  end
else
  compute new RRR for clustered eigenvalues
  goto **
end

Figure 1.5 Multiple Relatively Robust Representations (MRRR) algorithm.

MRRR algorithm usually does not require reorthogonalization to compute orthogonal eigenvectors corresponding to a group of clustered eigenvalues. In addition, each eigenvector can be computed independently, which enables a coarse-grained parallelization [7].

3) Back transform eigenvectors of the tridiagonal matrix to those of the original matrix through matrix multiplications.

Orthogonal transformation matrices can either be accumulated during the process of reduction, or constructed after the reduction has been completed. Given a group of Householder vectors $v_1, v_2, \ldots, v_k \in \mathbb{R}^n$, the corresponding orthogonal matrix of Householder transformations $H = H_1 H_2 \cdots H_k$ can be represented as $H = I - WY^T$ where $Y, W \in \mathbb{R}^{n \times k}$ [10], or as $H = I - YRY^T$ where $Y \in \mathbb{R}^{n \times k}$ and $R \in \mathbb{R}^{k \times k}$ is upper triangular [88]. LAPACK and ScaLAPACK use $H = I - YRY^T$ representation for back transformation, while SBR uses $H = I - WY^T$ representation.

All of the above algorithms successfully compute eigenpairs of real symmetric matrices to full accuracy. They have both sequential and parallel implementations. In efficient implementations, blocked algorithms are used whenever possible [1, 2, 13, 20]. Some other algorithms with inherent parallelism, such as the Homotopy method [22, 65, 67, 68, 73, 74] and the invariant subspace methods [54, 55, 5], have also attracted broad interest.

In our research, we pay attention not only to the blocked implementation of algorithms, but also to the blocked structure of the input matrix itself to reduce further the overhead of data access. Different algorithms are chosen based upon matrix structure and accuracy requirement provided by the user. The kernel of our approach is to parallelize the symmetric block tridiagonal divide-and-conquer (BD&C) eigensolver [42, 43], which computes approximate eigenpairs of a real symmetric block tridiagonal matrix directly, that is, not requiring any further reduction to a condensed form. Consequently, we handle input matrices according to the following classification:

1) Block tridiagonal matrices. A parallel BD&C eigensolver is implemented for the decomposition of such matrices. This parallel eigensolver computes the full spectrum of a real symmetric matrix up to a prescribed accuracy.

2) Dense matrices. For parallel eigen-decomposition of a dense matrix, ScaLAPACK subroutines can be used to compute eigenpairs to full accuracy efficiently. If lower accuracy is required, an alternative approach is likely to be more efficient. Earlier investigations [12, 11] have shown that reducing a full matrix to a banded matrix can be implemented using level 3 BLAS operations. By contrast, if we directly reduce a full matrix to a tridiagonal one [35], only half of the operations can exploit the high performance of level 3 BLAS operations. We extend this concept in that we first reduce a full matrix to a block tridiagonal form using orthogonal transformations, and then decompose the block tridiagonal matrix using the parallel BD&C eigensolver.

3) Sparse matrices. Reordering algorithms have been developed to reduce the bandwidth of an unstructured sparse matrix. Based on the permuted matrix, we may

determine a block tridiagonal structure on which we can apply the parallel BD&C eigensolver [4]. Note that the sparse matrix structure here can also be a dense matrix that is "effectively" sparse, meaning that although most of the matrix elements are nonzeros, many of them can be considered zero within the user-specified accuracy requirements of the eigenvalues. The concept of "effectively" sparse matrix is applicable to matrices with larger elements close to the diagonal and smaller elements away from the diagonal, which reflects a locality principle that frequently occurs in physical applications.

The parallel approximate eigensolver first determines what algorithm will be used to transform the matrix into block tridiagonal form depending on whether the input matrix has some structure or not. Then, the block tridiagonal matrix is decomposed using the parallel BD&C eigensolver.

1.4 General notation

Symbols that will be used consistently throughout this dissertation in all sections are listed on pages xv – xvi. Symbols used only in one specific section will be defined in their context when they are used.

Throughout this dissertation, matrices are denoted by uppercase letters. For example, A denotes a real symmetric matrix and A^T denotes the transpose of matrix A. The (i, j)-th element of matrix A will be represented by a_{ij}. A submatrix of A containing columns j_1 to j_2 and rows i_1 to i_2 will be denoted using the Matlab notation $A_{(i_1:i_2, j_1:j_2)}$. The j-th column of A will be denoted by a_j. A_{ij} will represent the (i, j)-th submatrix of A. The identity matrix will be denoted by I. The (i, j)-th element of I is given by δ_{ij}, with $\delta_{ij} = 0$, $\forall i \neq j$ and $\delta_{ii} = 1$.

Vectors are denoted by lower case letters such as v. The i-th element of v will be denoted by v_i. $e_k = \left(\underbrace{0,\ldots,0}_{k-1},1,\underbrace{0,\ldots,0}_{n-k} \right)^T$ will represent the vector with its k-th element having the value 1.

Lower case letters p, r and c will be used to denote the number of processors and the corresponding processor rows and columns in a processor grid. Letters m, n will be reserved for matrix sizes, and q will be used to denote the number of diagonal blocks of a block tridiagonal matrix.

Lower case Greek letters denote scalars. Eigenvalues of a real symmetric matrix of order n will be denoted by λ_1, λ_2, $\cdots$, etc., with $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_n$. Similarly, singular values of a matrix will be denoted by σ_1, σ_2, $\cdots$, σ_n, but sorted in descending order. The diagonal eigenvalue matrix will be denoted by Λ, while the singular value matrix by Σ.

A tilde over a symbol denotes the modified value of that quantity, while a circumflex over a symbol denotes a computed value. For example, $\tilde{A} = A + vu^T$ implies that $\tilde{A}$ is a rank-one update of A, and $\hat{\lambda}_i$ represents a computed approximate eigenvalue in contrast to an exact eigenvalue λ_i. $\lceil \alpha \rceil$ represents scalar α rounded up to the nearest integer, and $\lfloor \alpha \rfloor$ represents scalar α rounded down to the nearest integer.

$\mathbb{R}$ denotes the set of real numbers, and $\mathbb{R}^{m \times n}$ denotes the set of $m \times n$ real matrices. Unless explicitly specified otherwise, all matrices are of size $n \times n$ and all vectors are of size n.

Finally, since the terms “floating-point operations” and “floating-point operations per second” are used frequently to quantify computational complexity and performance of an implementation, respectively, we use *flops* to represent “floating-point operations”, and *FLOPS* to represent “floating-point operations per second”.

1.5 Outline of dissertation

This dissertation is organized as follows:

1) In Section 2, essential sequential algorithms for an approximate real symmetric eigensolver are reviewed. The block tridiagonal divide-and-conquer algorithm computes the full spectrum of a block tridiagonal matrix to prescribed accuracy. The orthogonal block tridiagonal reduction algorithm reduces a real symmetric dense matrix to block tridiagonal form using orthogonal transformations. The block tridiagonalization algorithm re-constructs an "effectively" sparse matrix into block tridiagonal form.

2) In Section 3, issues in design and implementation of parallel block tridiagonal divide-and-conquer eigensolver are discussed in detail. Analyses of complexities in computation and communication are given for understanding of performance and scalability.

3) In Section 4, parallel pre-processing algorithms of dense matrices and their implementations are presented. The purpose of those pre-processing steps is to construct a block tridiagonal matrix that is similar to the original dense matrix.

4) In Section 5, numerical results for the parallel approximate eigensolver and its major components are presented. A flow chart of major steps in the approximate eigensolver shows the criteria for choosing different algorithms depending on user specified requirements. Test matrices include those from applications in quantum chemistry and physics, random matrices, and matrices with specific eigenvalue distributions.

5) Finally, in Sections 6 and 7, we summarize results in this dissertation and discuss how some of our work can be further developed and improved.

2 Sequential algorithms for an approximate real symmetric eigensolver

As mentioned above, a parallel version of the block tridiagonal divide-and-conquer (BD&C) eigensolver with its ability to compute approximate eigensystems will be a key algorithm of our approximate eigensolver. We may also need pre-processing algorithms to obtain the necessary block tridiagonal structure if the input matrix does not initially possess such a structure. The sequential versions of these algorithms are reviewed below.

2.1 Block tridiagonal divide-and-conquer (BD&C) algorithm

Given a block tridiagonal matrix $M \in R^{n\times n}$ and an accuracy tolerance $\varepsilon_{mach} \le \tau < 0.1$ where ε_{mach} is the machine precision, the BD&C algorithm computes eigenpairs of M to the prescribed accuracy τ :

$$M = \begin{pmatrix} B_1 & C_1^T & & & \\ C_1 & B_2 & C_2^T & & \\ & C_2 & B_3 & \ddots & \\ & & \ddots & \ddots & C_{q-1}^T \\ & & & C_{q-1} & B_q \end{pmatrix} \approx V\Lambda V^T,$$

where q is the number of diagonal blocks, V is an approximation to the eigenvectors of M and Λ is a diagonal matrix containing approximations to the eigenvalues of M , so that $\left\| M - V\Lambda V^T \right\|_2 = O\left(\tau \left\| M \right\|_2\right)$ and V is numerically orthogonal.

There are three major steps in the BD&C algorithm [43]: subdivision, solution of subproblems and synthesis of solutions.

2.1.1 Subdivision

The off-diagonal blocks C_i of sizes $b_{i+1} \times b_i$ are approximated by lower rank matrices using their singular value decompositions:

$$C_i \approx \sum_{j=1}^{\rho_i} \sigma_j^i u_j^i v_j^{iT} = U_i \Sigma_i V_i^T ,$$

where ρ_i is the chosen approximate rank of C_i based on the accuracy requirement, $U_i \in \mathbb{R}^{b_{i+1} \times \rho_i}$ is the orthogonal matrix containing the first ρ_i left singular vectors, $V_i \in \mathbb{R}^{b_i \times \rho_i}$ contains the first ρ_i right singular vectors, Σ_i is the diagonal matrix that contains the largest ρ_i singular values of C_i, and $i = 1, 2, \cdots, q-1$.

Using the above factorizations, the block tridiagonal matrix M can now be represented as an updated block diagonal matrix as follows:

$$M = \tilde{M} + \sum_{i=1}^{q-1} W_i W_i^T, \tag{2.1}$$

where $\tilde{M} = diag\left\{\tilde{B}_1, \quad \tilde{B}_2, \quad \cdots, \quad \tilde{B}_q\right\}$,

$$\begin{aligned}
&\tilde{B}_1 = B_1 - V_1 \Sigma_1 V_1^T, \\
&\tilde{B}_i = B_i - U_{i-1} \Sigma_{i-1} U_{i-1}^T - V_i \Sigma_i V_i^T, \quad \text{for} \quad 2 \le i \le q-1, \\
&\tilde{B}_q = B_q - U_{q-1} \Sigma_{q-1} U_{q-1}^T,
\end{aligned}$$

$$W_1 = \begin{pmatrix} V_1 \Sigma_1^{1/2} \\ U_1 \Sigma_1^{1/2} \\ 0 \\ 0 \end{pmatrix}, \quad W_i = \begin{pmatrix} 0 \\ V_i \Sigma_i^{1/2} \\ U_i \Sigma_i^{1/2} \\ 0 \end{pmatrix} \text{ for } 2 \le i \le q-2, \text{ and } W_{q-1} = \begin{pmatrix} 0 \\ 0 \\ V_{q-1} \Sigma_{q-1}^{1/2} \\ U_{q-1} \Sigma_{q-1}^{1/2} \end{pmatrix}.$$

2.1.2 Solve subproblems

Each diagonal block $\tilde{B}_i$ is factorized:

$$\tilde{B}_i = Z_i D_i Z_i^T, \quad \text{for} \quad i = 1, 2, \cdots, q, \tag{2.2}$$

from which we obtain

$$\tilde{M} = Z D Z^T, \tag{2.3}$$

where

$Z = diag\{Z_1, Z_2, \cdots, Z_q\}$ is a block diagonal orthogonal matrix, and

$D = diag\{D_1, D_2, \cdots, D_q\}$ is a diagonal matrix.

Note that traditional algorithms may be applied to compute the eigen-decomposition of the diagonal blocks. Typically, the number of diagonal blocks q in a block tridiagonal matrix is much greater than 2 and the block sizes b_i are small compared to the matrix

size n. Thus, the eigen-decomposition of each subproblem $\tilde{B}_i$ in Equation 2.2 involves only a diagonal block, which yields better data access time pattern than traditional decomposition methods on the much larger full matrix.

2.1.3 Synthesis

From Equations 2.1 and 2.3 we have:

$$M = Z(D + \sum_{i=1}^{q-1} Y_i Y_i^T) Z^T , \tag{2.4}$$

where $Y_i = Z^T W_i$.

Denoting $S = D + \sum_{i=1}^{q-1} Y_i Y_i^T$ and $\rho = \sum_{i=1}^{q-1} \rho_i$ in the synthesis step, *S* is represented as a sequence of ρ rank-one modifications of *D*. The ρ_i rank-one modifications $D + y_j^i \left(y_j^i\right)^T$ $\left(j = 1, 2, \cdots, \rho_i \text{ and } i = 1, 2, \cdots, q-1\right)$ corresponding to an off-diagonal block C_i are called one merging operation, where $\left\{y_j^i\right\}$ are the vectors that determine Y_i. Thus, the algorithm performs a total of $q-1$ such merging operations. For each rank-one modification of the *i*-th merging operation, the modified matrix is first decomposed: $D + y_j^i \left(y_j^i\right)^T = V_j^i \Lambda_j^i \left(V_j^i\right)^T$, and the eigenvector matrix from this decomposition is then multiplied onto the accumulated eigenvector matrix starting with the block diagonal eigenvector matrix Z. The accumulation of an intermediate eigenvector matrix for each rank-one modification involves matrix-matrix multiplications. Figure 2.1 illustrates a possible merging sequence of a matrix with four blocks. The shaded areas are eigenvector matrix blocks.

Deflation happens when there is either a zero (or small) component in y_j^i or two equal (or close) elements in D [24, 36]. If the k-th component in y_j^i is zero, then the k-th diagonal d_k of D is an eigenvalue of $D + y_j^i \left(y_j^i\right)^T$ and the corresponding eigenvector is the identity vector e_k. If there are two equal elements on the diagonal of D, Givens

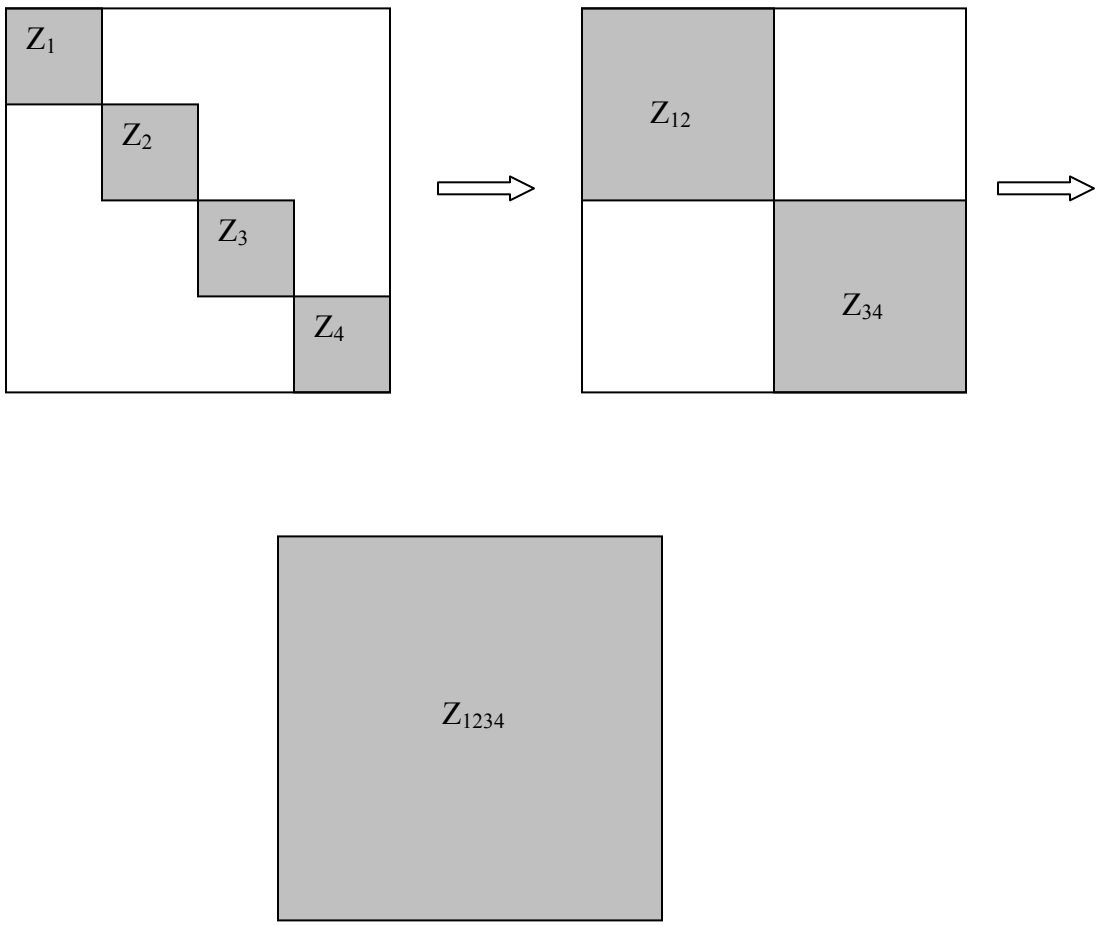

Figure 2.1 Merging operations to accumulate eigenvectors.

rotation is used to zero out one of the corresponding element in y_j^i, and corresponding eigenpairs can be computed as the former case. When deflation occurs, no computation is required to compute and accumulate the corresponding eigenvector. Further, a permutation matrix P is used to move the deflated components of y_j^i to the bottom of y_j^i:

$$M = ZG^T P^T \hat{P}^T \hat{P} P G (D + \sum_{i=1}^{q-1} Y_i Y_i^T) G^T P^T \hat{P}^T \hat{P} P G Z^T,$$

so that columns in $\hat{Z} = ZG^T P^T$ are re-grouped according to their structure [50, 92]. The structure of $\tilde{Z} = ZG^T P^T \hat{P}^T$ from the first rank-one modification of a merging operation is shown in Figure 2.2, and that from the rank-one modifications after the first one is shown in Figure 2.3.

The deflation criteria can be relaxed if the accuracy tolerance is greater than full accuracy. Under this condition, the synthesis step also involves approximations. As shown in Figure 2.4 [43], the percentage of deflation increases drastically as the blocks in the accumulated Z matrix become larger.

Moreover, the approximate rank of the off-diagonal blocks in the first step of BD&C typically becomes smaller as the accuracy requirement becomes lower, which also reduces the computational complexity. Those two factors lead to high efficiency of the BD&C algorithm as accuracy decreases as demonstrated on a random block tridiagonal matrix in Figure 2.5 [43].

A merging operation is a balanced one if the sizes b_1 and b_2 of the two blocks to be merged are approximately the same, i.e., $b_1 \approx b_2$. If $b_1 \gg b_2$ or $b_1 \ll b_2$, then the merging operation is an unbalanced one. It has been shown that the time complexity for the most unbalanced merging operation is less than that for the most balanced one but with a higher rank – even an increase in rank of only one [43]. Therefore, a block tridiagonal structure is preferred that allows for low rank modifications in the final merging operation, regardless of the relative sizes of the blocks being merged. In our parallel approximate eigensolver, advantage is taken of this fact whenever possible.

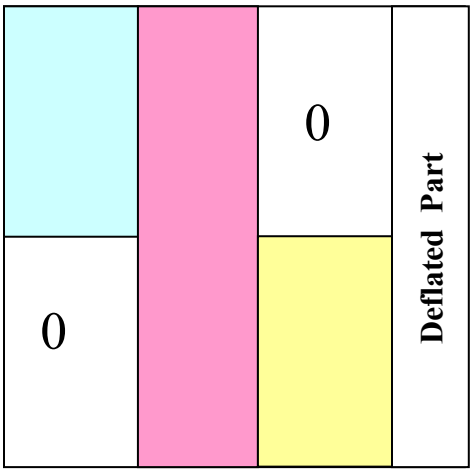

Figure 2.2 Structure of $\tilde{Z}$ from the first rank-one modification in a merging operation.

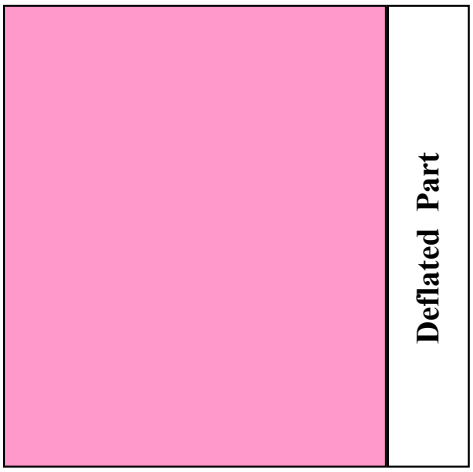

Figure 2.3 Structure of $\tilde{Z}$ from rank-one modifications after the first one in a merging operation.

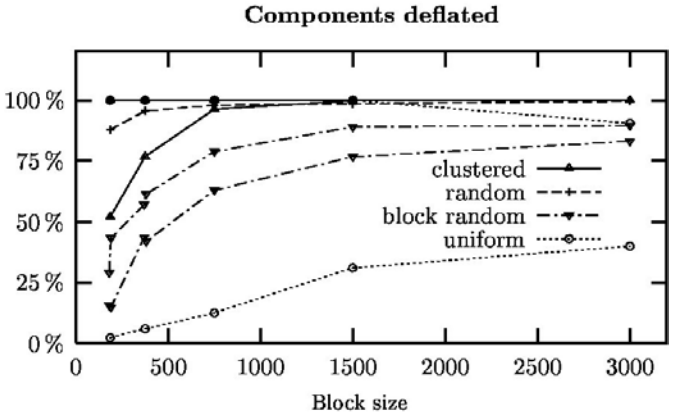

Figure 2.4 Lower and upper bound for deflation in the merging operations with different types of eigenvalue distribution. Matrix size $n = 3000$ with constant block size $b = 10$ and $\tau = 10^{-4}$ [43].

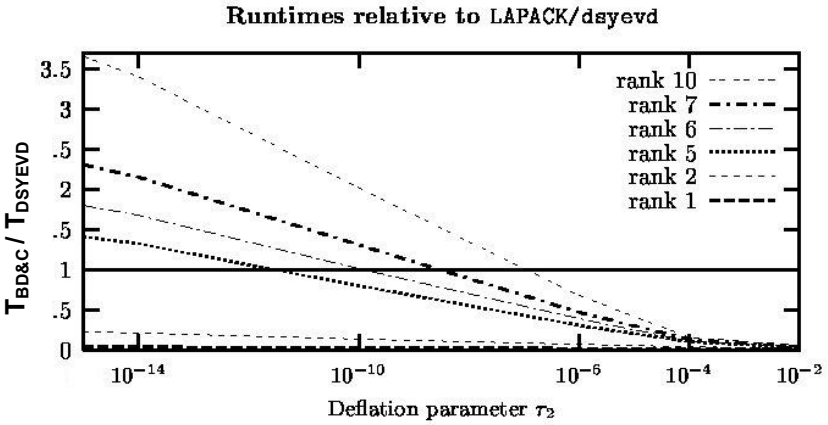

Figure 2.5 Execution time with different deflation tolerances and ranks, matrix size $n = 3000$ with constant block size $b = 10$ [43].

2.1.4 Computational complexity of BD&C

Assume that matrix $M \in \mathbb{R}^{n\times n}$ is a real symmetric block tridiagonal matrix with q diagonal blocks, n is divisible by q and each block has the same size $b = n/q$. To simplify the time complexity analysis, we further assume that each off-diagonal block has the same rank ρ.

For the BD&C algorithm, if deflation is not counted, the dominant part of the computational time is the matrix multiplications to accumulate eigenvectors during the merging operations; the complexity of all other computations, i.e. solving secular equations, computing eigenvectors, is $O(n^2)$ or less. Therefore, the leading term in the computational complexity of BD&C (i.e., matrix multiplications) is

$$\begin{aligned} flops_{BD\&C} &= \sum_{i=0}^{\lceil \log q \rceil - 1} \left(\rho - \frac{1}{2}\right) 2n^3 \left(\frac{1}{4}\right)^i \\ &= \frac{8}{3}\left(\rho - \frac{1}{2}\right) n^3 \left(1 - \frac{1}{q^2}\right) \\ &\approx \frac{8}{3}\rho n^3 - \frac{4}{3}n^3 + O\left(\rho b^2 n\right) \end{aligned} \tag{2.5}$$

2.2 Transformation of "effectively" sparse matrix – block tridiagonalization (BT) algorithm

Most matrices generated in real applications do not have a block tridiagonal structure; however, many may be sufficiently approximated by one. Given a full symmetric matrix $A \in R^{n\times n}$ and an accuracy tolerance $\varepsilon_{mach} \le \tau < 0.1$, A is called "effectively" sparse if many of the nonzeros of A may be set to zero without perturbing the eigenvalues of A more than $\tau \|A\|$. The 6-step heuristic Block Tridiagonalization (BT) algorithm [4] has been developed to transform a full matrix that is "effectively" sparse into a sparse matrix and then find a block tridiagonal structure for the sparse matrix as shown in Figure 2.6.

$$A=\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & \cdots & a_{3n} \\ a_{41} & a_{42} & a_{43} & a_{44} & \cdots & a_{4n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & \cdots & a_{nn} \end{pmatrix} \Rightarrow \begin{pmatrix} a_{11} & 0 & a_{13} & 0 & \cdots & 0 \\ 0 & a_{22} & a_{23} & a_{24} & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & 0 & \cdots & a_{3n} \\ 0 & a_{42} & 0 & a_{44} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & a_{n3} & 0 & \cdots & a_{nn} \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} B_1 & C_1^T & & & \\ C_1 & B_2 & C_2^T & & \\ & C_2 & B_3 & \ddots & \\ & & \ddots & \ddots & C_{q-1}^T \\ & & & C_{q-1} & B_q \end{pmatrix} = M$$

Figure 2.6 Transform a full symmetric matrix into a block tridiagonal matrix [4].

The BT algorithm partitions τ into two parts, $\tau = \tau_1 + \tau_2$, allowing a portion of the acceptable error to be used for different steps in the algorithm. The algorithm is described below.

2.2.1 The 6-step block tridiagonalization algorithm

Step 1. Global threshold A with $\sqrt{\tau}\|A\|$

We start with a threshold $\tau' = \sqrt{\tau}$, larger than permitted by the accuracy requirement, and obtain matrix A' by eliminating all elements in A less than $\sqrt{\tau}\|A\|$. For many matrices resulting from modeling physical phenomena with strong locality properties, most of the elements will be eliminated. The resultant matrix A' will contain only the largest elements of A and would hopefully be sparse. We start with a randomly permuted matrix shown in Figure 2.7 as an example. Figure 2.8 shows A' as the resultant matrix from a global threshold of A. The vertical color bar to the right of the matrix indicates the magnitudes of the matrix elements by color; that is, matrix elements whose

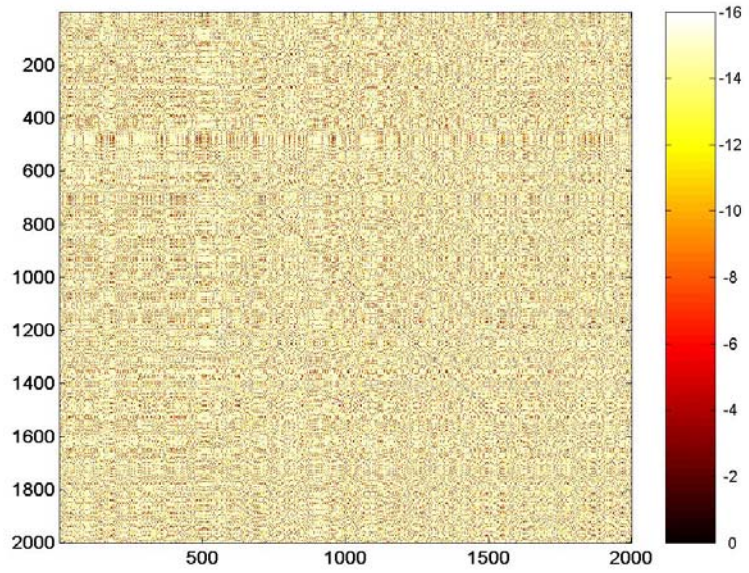

Figure 2.7 A randomly permuted matrix A.

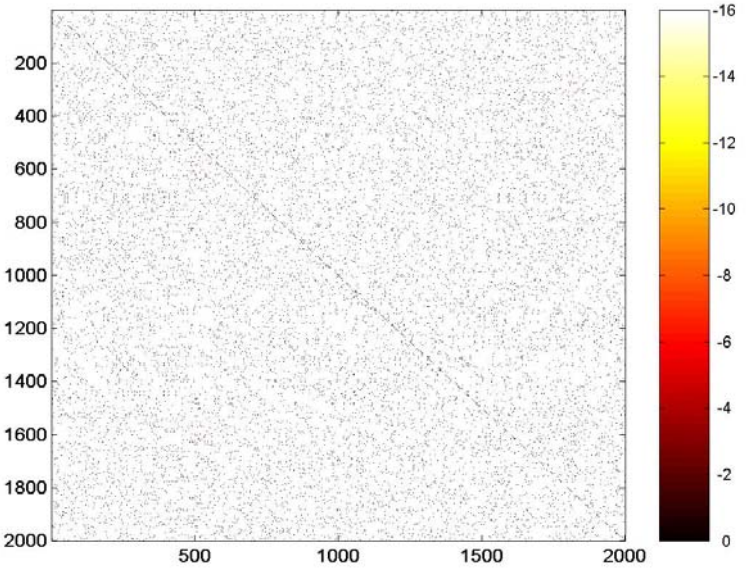

Figure 2.8 A' from global threshold of A, $\tau = 10^{-6}$.

magnitudes are of order 1 or larger are essentially black, while elements with smaller magnitudes go from black to red, then to yellow and finally to white.

Step 2. Reorder A'

In this step, A' is reordered to reduce its bandwidth using the Gibbs-Poole-Stockmeyer (GPS) algorithm [46, 64, 23]. Thus, the elements of A' are moved closer to the diagonal. Figure 2.9 shows that the bandwidth of A' has been greatly reduced after the permutation. The permutation matrix *P* accomplishing this task is obtained and will be used in Step 3.

Step 3. Permute A ***with permutation matrix*** P ***from Step 2***

The permutation matrix *P* computed in Step 2 is applied to *A*, resulting in matrix $A'' = P^T AP$. The larger elements of *A* are expected to be closer to the diagonal in A'' as shown in Figure 2.10.

Step 4. Target threshold A'' ***with*** $\tau_1 \|A\|$

In this step, we try to eliminate those elements far away from the diagonal in matrix A'' whose influence on the error of any eigenvalue is negligible compared to $\tau_1 \|A\|$. This step produces matrix A''' such that

$$A'' = A''' + E, \quad \text{with } \|E\|_1 < \tau_1 \|A\|.$$

It can be shown [26, 49] that the absolute difference between the eigenvalues λ_i of A'' and the eigenvalues λ_i' of A''' is bounded by

$$|\lambda_i - \lambda_i'| \le \|E\|_2 \le \|E\|_1 .$$

Since the eigenvalue errors are bounded by the 1-norm of the error matrix *E*, the algorithm traverses the matrix elements along the off-diagonals from the end toward the center as illustrated in Figure 2.11, zeroing elements before each column-wise sum of absolute values of the dropped elements exceeds $\tau_1 \|A\|$. Figure 2.12 shows A''' as the result of target threshold of A''.

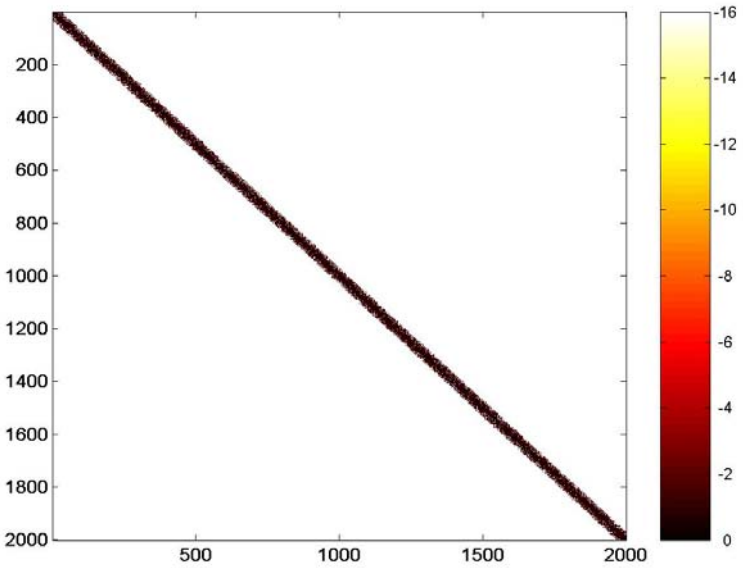

Figure 2.9 Permuted A' using the GPS algorithm.

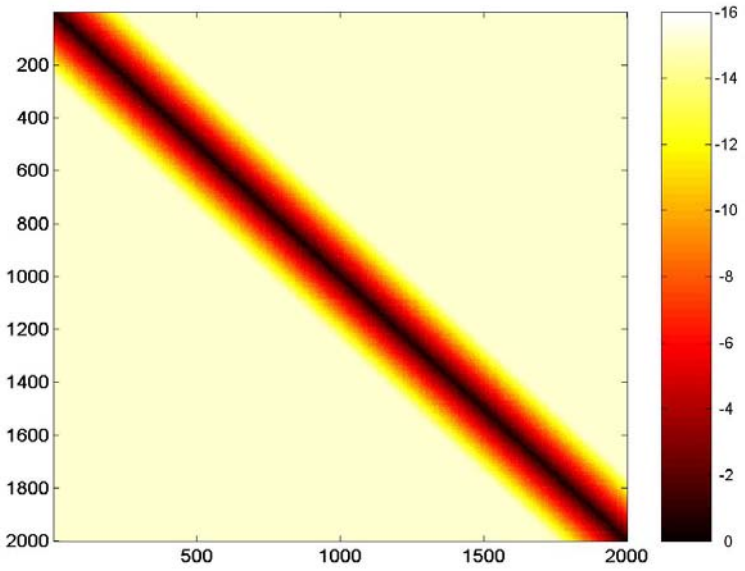

Figure 2.10 Permuted $A'' = P^T AP$.

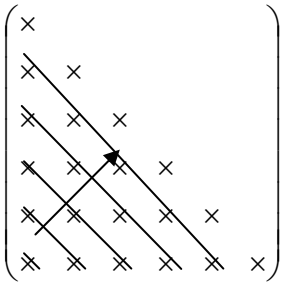

Figure 2.11 Traverse elements along matrix off-diagonals [4].

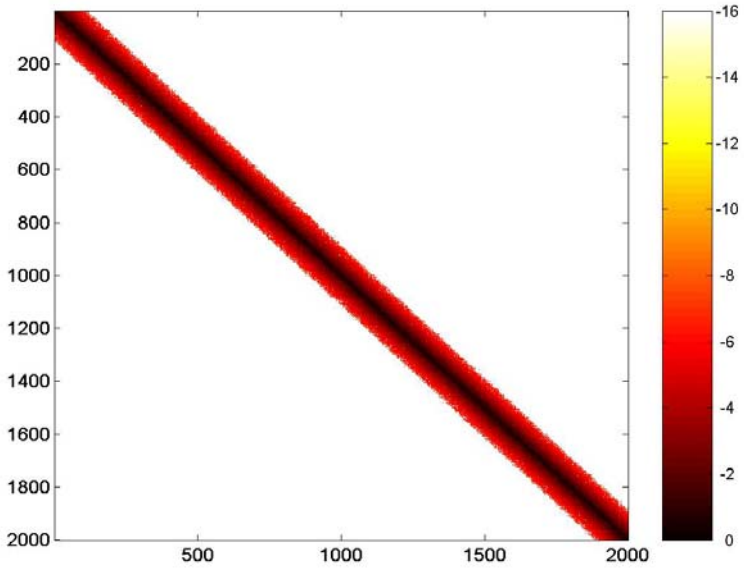

Figure 2.12 A'' after target threshold, $\tau_1 = 10^{-6}$.

Step 5. Covering A'''

The sizes of the diagonal blocks, which also fix the sizes of the off-diagonal blocks, are determined such that the resulting block tridiagonal matrix contains all the matrix elements that are effectively nonzero (i.e., nonzeros in A'''). These are the matrix elements whose effect on the accuracy of the eigenpair approximation may be non-negligible. Figure 2.13 shows block sizes obtained from A''' along x-axis.

Step 6. Target block reduction (TBR)

As an option, the last step of the BT algorithm attempts to produce a few small blocks for a lower computational complexity in the merging operations of the BD&C algorithm.

In step 4, none of the matrix elements dropped are greater than the given error bound $\tau_1 \|A\|$. It may be possible to eliminate some of the matrix elements whose absolute values are larger than the given error bound without causing the accumulative error in the eigenvalues to exceed this error. Wilkinson [96] has given a sensitivity analysis that estimates the eigenvalues of a perturbed matrix $M + \varepsilon E$ in terms of the eigenvalues and eigenvectors of the original matrix *M*:

$$\lambda(M + \varepsilon E) = \lambda(M) + \varepsilon(x^T E x) + O(\varepsilon^2), \tag{2.6}$$

where *x* denotes the eigenvector corresponding to the eigenvalue $\lambda(M)$ of M.

From Equation 2.6, the eigenvalue error as a result of zeroing matrix elements m_{ij} and m_{ji} can be estimated by

$$\Delta\lambda = 2m_{ij}x_i x_j + O(m_{ij}^2). \tag{2.7}$$

Several elements may be eliminated as long as the maximum of the sum of the eigenvalue errors is less than the given error bound. In our case, this error bound is $\tau_2 \|A\|$. Note that step 6 is only possible if an approximation for the eigenvectors is available. For an iterative method solving a non-linear eigenvalue problem (like the SCF method), we may use the eigenvectors from the previous iteration as an approximation. There may be other similar applications with eigenvector approximations permitting this last step in the algorithm.

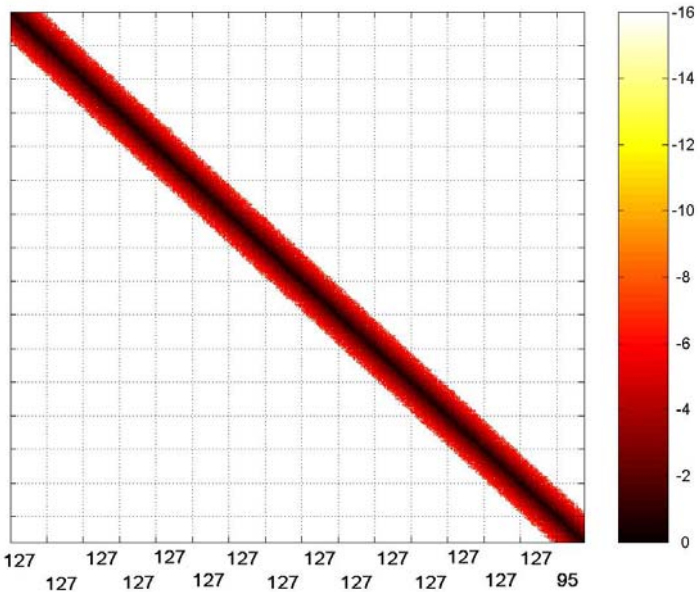

Figure 2.13 Blocks that cover all nonzeros of A'''.

Using TBR, we may reduce the size of a few diagonal blocks and hope that the corresponding off-diagonal blocks have a lower rank. As an example, for a matrix $M \in R^{8x8}$ as shown in Figure 2.14, eliminating elements m_{25}, m_{35}, and their symmetric counterpart m_{52} and m_{53} would lead to a totally different block tridiagonal structure as illustrated in Figure 2.15.

2.2.2 Computational complexity of BT

Most of the operations involved in the BT algorithm are comparisons, additions and permutations. The computational complexity and the number of data accessed are both $O(n^2)$. Let nnz_1 and nnz_2 be the number of nonzero elements of matrices A' and A''', respectively (typically $nnz_1 < nnz_2 \ll n$). In Table 2.1, the maximal time complexity for each step of the algorithm is listed. In Step 6, *k* denotes the number of matrix elements that are checked for elimination (typically $k << n$). Since nnz_1 and nnz_2 are both no greater than n^2, total complexity of steps 1 – 5 of BT is $O(n^2)$ regardless of their values.

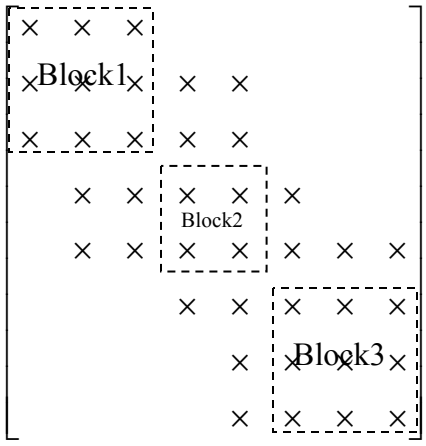

Figure 2.14 Block tridiagonal structure that covers all nonzeros.

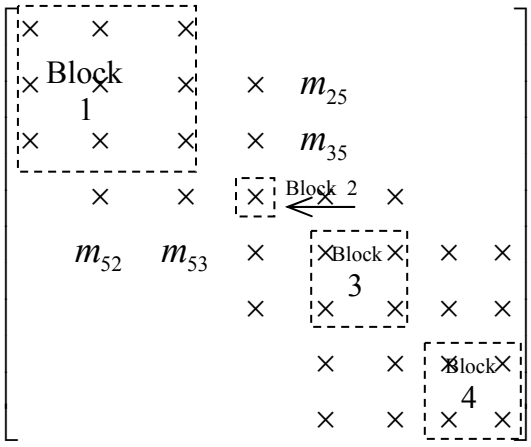

Figure 2.15 Block tridiagonal structure after eliminating entries (2,5), (3,5), (5,2) and (5,3).

Table 2.1 Worst-case time complexity of BT [4].

Steps	Comparison and data movement	Addition and multiplication
1. Global threshold	n^2	—
2. GPS reorder	$\frac{3}{2}n^{\frac{3}{2}}$ [45]	—
3. Symmetric permutation	$3n^2$	—
4. Target threshold	$n^2 - nnz_2$	$n^2 - nnz_2$
5. Covering	n	—
6. Reduce block size	$2n$	$2kn$

However, the bound on k is n^2, so the complexity of step 6 could be $O(n^3)$. The computational complexity of the BT algorithm is $O(n^2)$ when $k \le O(n)$.

2.3 Orthogonal block tridiagonal reduction of dense matrix (OBR)

If a full symmetric matrix cannot be transformed into a block tridiagonal matrix for use by the BD&C algorithm with little computational effort as described above, one may choose to use a sequential eigensolver from a robust and efficient numerical library (e.g. DSYEVD [87] from LAPACK) to decompose it. However, for large matrices, the data locality in the reduction-to-tridiagonal step may not be as good as those of matrices of moderate size. Studies have shown that by reducing the dense matrix successively to a banded matrix and finally tridiagonal matrix [12, 41], one has a better data access pattern and larger portion of level 3 BLAS operations. We further extend this idea to produce a sequential algorithm for the reduction to block tridiagonal form.

Given a dense real symmetric matrix $A \in \mathbb{R}^{n \times n}$, we desire to apply a sequence of orthogonal similarity transformations to reduce A to a block tridiagonal matrix M.

There are different ways to construct the orthogonal matrices, for instance, the QR factorization and the singular value decomposition. This section will consider only the QR factorization algorithm.

2.3.1 Reduction using QR factorization

The orthogonal transformations annihilate elements below the block subdiagonal panel by panel as shown in Figure 2.16. We will denote each matrix panel by $G_i \in \mathbb{R}^{m_i \times n_i}$, each diagonal block of M by B_i and off-diagonal block by C_i. As a general rule, each matrix panel has same panel width p_b, and all diagonal blocks have the same size $b_1 = b_2 = \cdots = b_{q-1} = b$ where q is the number of diagonal blocks and $q = \lfloor (n-1)/b \rfloor + 1$, except that the last block has the size $b_q = n - (q-1)b$.

We start with $A_0 = A$, QR factorization of the first panel $G_1 = Q_1 R_1$ is computed, and we obtain the first diagonal block B_1 and off-diagonal block C_1 which is the upper triangular part of R_1 as shown in Figure 2.17. Submatrix A_1 is updated using $Q_1^T A_1 Q_1$. Next the second panel G_2 is factorized into $Q_2 R_2$. Then we obtain blocks B_2 and C_2, and update submatrix A_2 in the same way as shown in Figure 2.18.

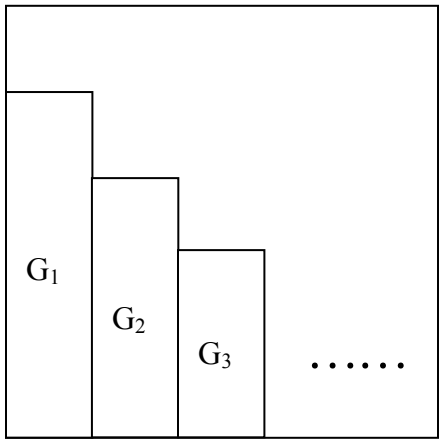

Figure 2.16 Orthogonal factorization performed in column blocks.

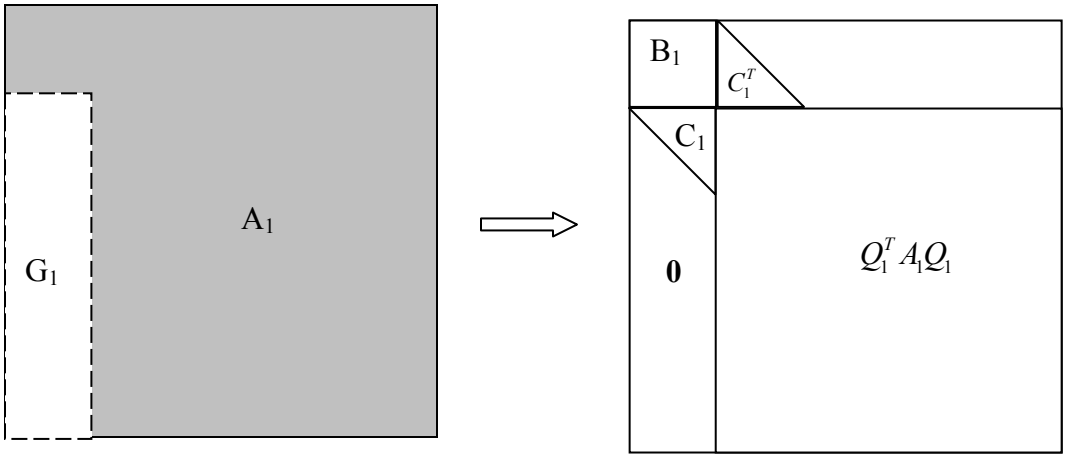

Figure 2.17 Reduction of the first panel.

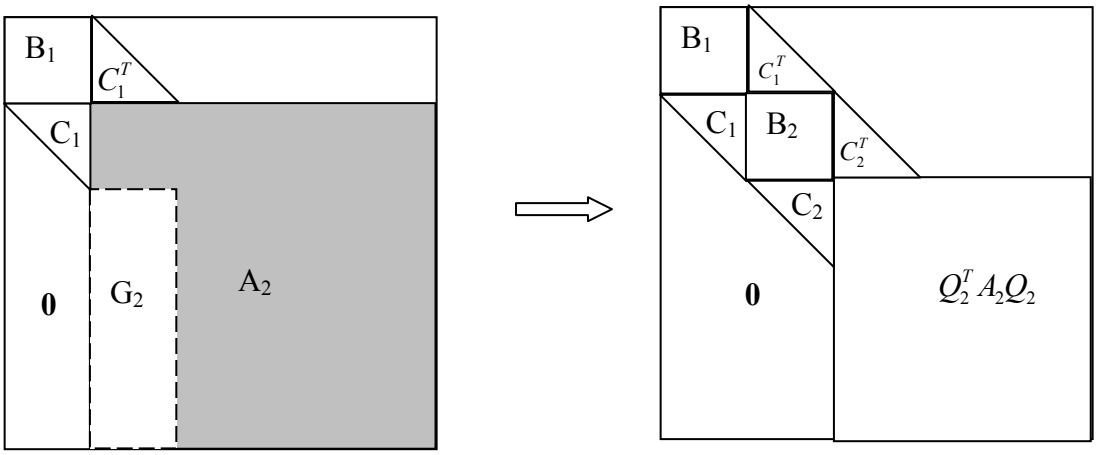

Figure 2.18 Reduction of the second panel.

In a general case, p_b is the width and $m_i = n - i \times p_b$ is the length of the *i*-th panel G_i, b is the block size of the reduced block tridiagonal matrix. In this Section and Section 2.3.2, we only illustrate the case when $p_b = b$. The cases of $p_b \neq b$ are discussed in Section 2.3.3.

Let $A_i \in \mathbb{R}^{(m_i+n_i)\times(m_i+n_i)}$ as illustrated by Figure 2.19 be the lower right principal submatrix of A at the *i*-th stage of orthogonal reduction. For each matrix panel $G_i \in \mathbb{R}^{m_i \times n_i}$ in Figures 2.16 and 2.19, its QR factorization $G_i = Q_i R_i$ where $Q_i \in \mathbb{R}^{m_i \times m_i}$ and $R_i \in \mathbb{R}^{m_i \times n_i}$ is used to reduce A to a block tridiagonal matrix. Partition $R_i = \begin{bmatrix} \hat{R}_i \\ \\ 0 \end{bmatrix}$ where $\hat{R}_i = (R_i)_{(1:n_i,1:n_i)}$ is upper triangular. A will be reduced to a block tridiagonal matrix with triangular off-diagonal blocks. Partitioning A_i as

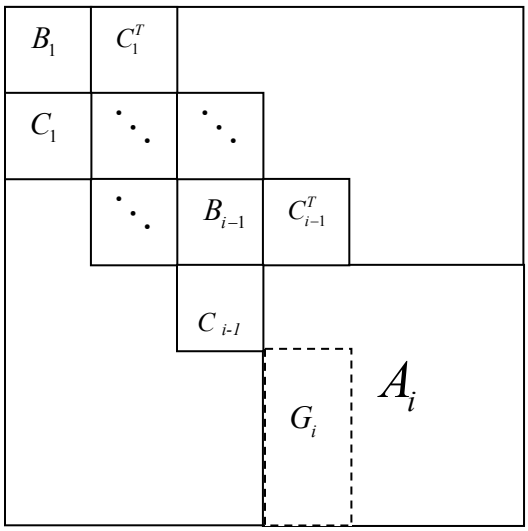

Figure 2.19 Matrix A at the *i*-th stage of orthogonal reduction.

$$A_i = \begin{array}{c} \begin{matrix} n_i & m_i \end{matrix} \\ \begin{bmatrix} A_{11}^i & A_{12}^i \\ A_{21}^i & A_{22}^i \end{bmatrix} \begin{matrix} n_i \\ m_i \end{matrix} \end{array},$$

and then applying Q_i to it, we have

$$\begin{bmatrix} I & O \\ O & Q_i^T \end{bmatrix} A_i \begin{bmatrix} I & O \\ O & Q_i \end{bmatrix} = \begin{bmatrix} A_{11}^i & R_i^T \\ R_i & \tilde{A}_{22}^i \end{bmatrix}. \tag{2.8}$$

The diagonal block A_{11}^i and off-diagonal block $\widehat{R}_i$ can be obtained directly, and $A_{i+1} = \tilde{A}_{22}^i = Q_i^T A_{22}^i Q_i$. We continue this procedure until the whole matrix A is reduced to a block tridiagonal matrix M. All the subdiagonal blocks of M except the last one are upper triangular. The panel width p_b needs to be chosen carefully. It should be small enough to keep cache miss rate low and yet large enough to benefit from data-reuse in level 3 BLAS operations.

2.3.2 Computational complexity of OBR

To reduce a real symmetric matrix A to a block tridiagonal matrix M with fixed panel width p_b and block sizes $b = p_b$ as shown in Figure 2.19, computational complexity of QR factorization $G_i = Q_i R_i$ of each matrix column block $G_i \in \mathbb{R}^{m_i \times n_i}$ where $n_i = p_b$ and $m_i = n - i \times p_b$ is $2n_i^2 m_i - \frac{2}{3} n_i^3$ [49]. Here the Householder vectors are saved for the update of A_{22}^i and the computational complexity to construct Q_i is also $2n_i^2 m_i - \frac{2}{3} n_i^3$ [49]. Finally, the time complexity for rank-2b updating of A_{22} is approximately $4m_i^2 n_i$ [49]. Assume that the size n of A is divisible by p_b and $q_b = n / p_b$ is the number of panels in A. The total number of floating-point operations for reduction from A to M thus becomes

$$
\begin{aligned}
flops_{OBR} &= \sum_{i=1}^{q_b-1}\left[4\left(q_b - i\right) p_b^3 - \frac{4}{3} p_b^3 + 4\left(q_b - i\right)^2 p_b^3 \right] \\
&= p_b^3 \left[2q_b\left(q_b - 1\right) - \frac{4}{3}(q_b - 1) + \frac{2\left(q_b - 1\right) q_b \left(2q_b - 1\right)}{3} \right] \\
&= p_b^3 \left(\frac{4}{3} q_b^3 - \frac{8}{3} q_b + \frac{4}{3} \right) \\
&\approx \frac{4}{3} n^3 - \frac{8}{3} p_b^2 n + O\left(p_b^3\right), \qquad (2.9)
\end{aligned}
$$

and the number of floating-point operations in rank-2*b* update that are level 3 BLAS equals

$$
flops_{OBR}^{BLAS3} = \sum_{i=1}^{q_b-1}\left[4\left(q_b - i\right)^2 p_b^3 \right] = \frac{4}{3} n^3 - 2 p_b n^2 + \frac{2}{3} p_b^2 n \,. \qquad (2.10)
$$

Although the leading term of computational complexity of both ORB and LAPACK tridiagonal reduction is $\frac{4}{3}n^3$ [35], the performance of OBR should be better due to higher ratio of level 3 BLAS operations. This is confirmed by the performance test results shown in Figure 2.20. In this test, we use subroutine DSYRDB from the SBR package [8] to reduce a real symmetric matrix to a block tridiagonal matrix with two different block sizes, 32 and 64, and use subroutine DSYTRD from LAPACK [1, 2] to reduce the same matrix to tridiagonal form. The panel width in DSYRDB equals the block size. The processor we use is one of the thirty-two 1.3 GHz Power4 processors on a node of the IBM p690 system at Oak Ridge National Laboratory [98]. The system has 27 nodes, and most of the nodes have 32 GB of memory. Level-1 instruction cache is 64 KB per processor, and the data cache is 32 KB per processor. The level-2 cache is 1.5 MB shared between the two processors. The level 3 cache is 32 MB and is off chip. The system goes by the nickname Cheetah at ORNL.

In Figure 2.20, the ratios of execution times and floating-point instructions (FLPINS) measured by PAPI [14, 15] show that the difference between execution time is much greater than that between floating-point operation count, and larger block size brings slightly better performance in general.

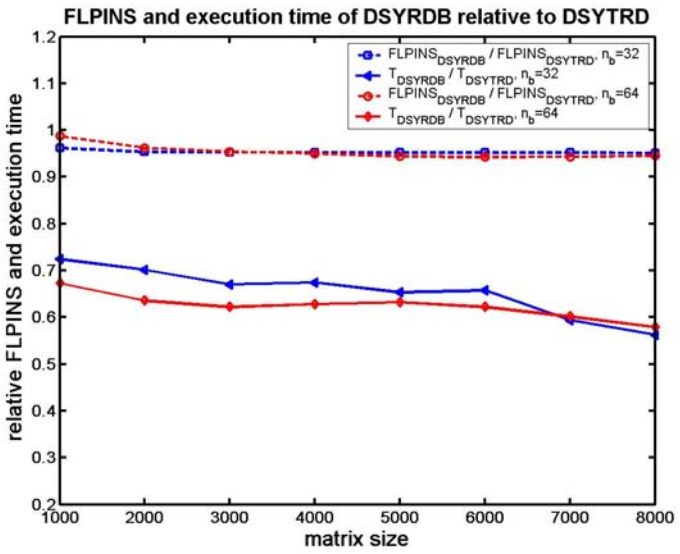

Figure 2.20 Ratio of execution time and FLPINS of DSYRDB to DSYTRD.

We may use QR factorizations to reduce matrix A to a block tridiagonal matrix M, and then use BD&C to compute the eigenpairs of M. Adding Equations 2.5 and 2.9 yields the total computational complexity of eigen-decomposition of a dense matrix using OBR followed by BD&C without deflation:

$$flops_{FULL} \approx \frac{8}{3}\rho n^3 + O(b^2 n)\,. \qquad (2.11)$$

In contrast, the time complexity for first reducing matrix A to a tridiagonal matrix T and then computing the eigenpairs of T using the divide-and-conquer method with no deflation adds up to $\frac{8}{3}n^3 + O\left(n^2\right)$, which is lower than the combination of OBR and BD&C unless $\rho = 1$. However, taking into account improved performance of DSYRDB over DSYTRD, the consequent higher ratio of deflation (see Figures 2.4 and 2.5) for lowered accuracy requirements, and a better data access pattern, the performance of the former algorithm may not necessarily be worse than the latter one.

2.3.3 Relationship between panel width and block size

In the block tridiagonal reduction algorithm in Section 2.3.1, there are two closely related parameters: the panel width p_b for the blocked QR factorization and the block size b for the block tridiagonal matrix. There are three possible combinations for p_b and b: $p_b > b$, $p_b < b$ and $p_b = b$.

We first consider the most straightforward case $p_b = b$ shown in Figure 2.21. The reduction algorithm involves QR factorizations of column blocks $G_i \in \mathbb{R}^{m_i \times n_i}$, accumulation of Householder transformations in blocked form $Q_i = I - Y_i W_i^T$ where Y_i contains columns of Householder vectors and W_i contains columns of scaled Householder vectors, construction of the update matrix

$$Z_i = A_i W_i - \frac{1}{2} Y_i W_i^T A_i W_i, \tag{2.12}$$

and update of the submatrix of A_i (yellow shade) with rank-$2p_b$ updates

$$A_{i+1} = A_i - Y_i Z_i^T - Z_i Y_i^T. \tag{2.13}$$

Next we consider the case of $p_b < b$. As shown in Figure 2.22, after the QR factorization of panel G_i, a one-side update of $b - p_b$ columns (gray shade part) must be computed, then the block Householder transformations $Q_i = I - Y_i W_i^T$ can be applied to submatrix of A_i from both sides as in the case of $p_b = b$.

In the last case $p_b > b$ as shown in Figure 2.23, the update of $p_b - b$ columns in gray shade by the Householder transformations from the right involves accessing all entries of submatrix A_i; while in the cases of $p_b < b$ and $p_b = b$, reduction of each G_i only requires accessing data in G_i. Therefore, reduction of G_i with $p_b > b$ cannot be computed directly by QR factorization without accessing the matrix entries outside G_i.

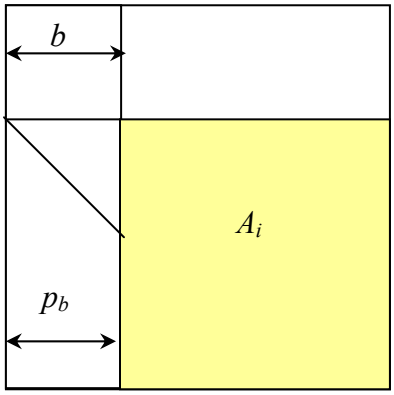

Figure 2.21 Orthogonal reduction in the case of $p_b = b$.

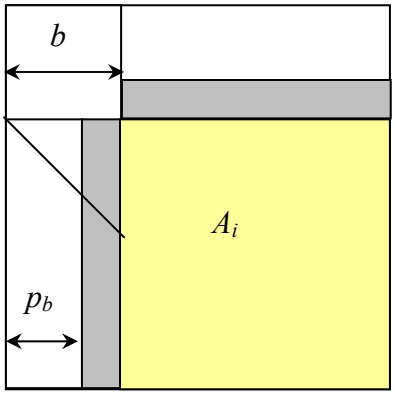

Figure 2.22 Orthogonal reduction in the case of $p_b < b$.

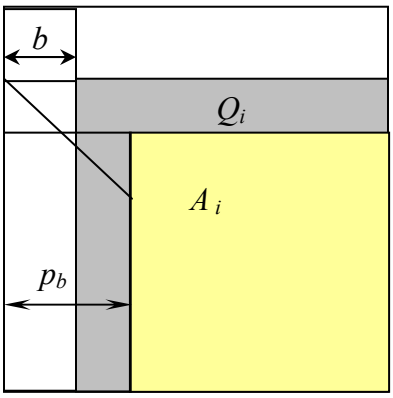

Figure 2.23 Orthogonal reduction in the case of $p_b > b$.

Consequently, matrix Z_i for the rank-$2p_b$ update used in Equation 2.13 cannot be computed using matrix multiplications as shown in Equation 2.12. However, Z_i can be constructed column by column using the formula

$$z_j^i = A_{j-1}^i w_j^i - \frac{1}{2} y_j^i \left(w_j^i \right)^T A_{j-1}^i w_j^i, \quad 1 \le j \le p_b \qquad (2.14)$$

during each Householder transformation. This approach is similar to the LAPACK symmetric tridiagonal reduction subroutine DSYTRD.

Theorem 2.1 For $n \ge 4b$ where n is the matrix size and b is the block size in the orthogonal block tridiagonal reduction algorithm that uses QR factorizations, a ratio of level 3 BLAS operations greater than 50% can be obtained only if the algorithmic panel width p_b is no greater than the block size b.

Proof. Based on the above observation of three cases of p_b and b, we can estimate how many level 3 BLAS operations are exploited in each case. Here we still assume that the panel width p_b is divisible by the matrix size n and that $q_b = n / p_b$ is the total number of matrix column blocks.

Case 1) $p_b = b$. Based on Equations 2.9 and 2.10, the ratio of level 3 BLAS operation

$$ratio_{p_b=b}\left(BLAS\ 3\right) = \frac{\frac{4}{3}n^3 - 2p_b n^2 + \frac{2}{3}p_b^2 n}{\frac{4}{3}n^3 - \frac{8}{3}p_b^2 n}$$

$$> \frac{\frac{4}{3}n^3 - 2p_b n^2}{\frac{4}{3}n^3}$$

$$= 1 - \frac{3p_b}{2n}$$

$$= 1 - \frac{3b}{2n}. \qquad (2.15)$$

Case 2) $p_b < b$. The floating-point operation count for each one-side update of $b - p_b$ columns as shown in gray shade in Figure 2.22 is $(4p_b + 1)(n - p_b i)(b - p_b)$ and in total takes

$$flops_{one-side} = \sum_{i=1}^{q_b - 1} (4p_b + 1)\, p_b\, (q_b - i)(b - p_b)$$

$$= (b - p_b)\left[\left(2 + \frac{1}{2p_b}\right)n^2 - n\left(2p_b + \frac{1}{2}\right)\right]$$

floating-point operations.

When accumulated Householder transformations are applied to the $b - p_b$ columns from the left, $b - p_b$ must be large enough so that this one-sided update can exploit the high performance of level 3 BLAS operations. Otherwise, we do not expect the high performance of level 3 BLAS operations can be fully exploited. Consequently, the ratio of level 3 BLAS operation satisfies the following inequality:

$$ratio_{p_b < b}(BLAS\ 3) \geq \frac{\frac{4}{3}n^3 - 2p_b n^2 + \frac{2}{3}p_b^2 n}{\frac{4}{3}n^3 - \frac{8}{3}p_b^2 n + (b - p_b)\left[\left(2 + \frac{1}{2p_b}\right)n^2 - n\left(2p_b + \frac{1}{2}\right)\right]}$$

$$> \frac{\frac{4}{3}n^3 - 2p_b n^2}{\frac{4}{3}n^3 + \frac{5}{2}(b - p_b)n^2}$$

$$= 1 - \frac{5b - p_b}{\frac{8}{3}n + 5(b - p_b)}$$

$$\geq 1 - \frac{15b}{8n}. \qquad (2.16)$$

Case 3) $p_b > b$. The dominant computational cost of each vector z_j^i using Equation 2.13 is a matrix-vector multiplication and takes $2\left(n - p_b i - j + 1\right)^2$ floating-point operations, $1 \le i < q_b$ and $1 \le j \le p_b$. In total, the construction of vectors z_j^i takes

$$flops_{construct\_z} = \sum_{i=1}^{q_b - 1} \sum_{j=1}^{p_b} 2\left(n - p_b i - j + 1\right)^2$$

$$= \sum_{k=p_b+1}^{n-p_b} 2k^2$$

$$= \frac{2}{3}\left(n - p + 1\right)^3 - \left(n - p + 1\right)^2 + \frac{n}{3} - \frac{2p}{3} - \frac{2}{3}\left(p + 1\right)^3 + \left(p + 1\right)^2$$

$$= \frac{2}{3}n^3 - \left(2p_b - 1\right)n^2 + \left(2p_b^2 - 2p_b + \frac{1}{3}\right)n + O\left(p^3\right).$$

Since the leading term of the computational complexity of orthogonal reduction is $\frac{4}{3}n^3$, approximately 50% of the floating-point operations are level 2 BLAS operations to compute z_j^i when $p_b > b$, similar to that of the LAPACK tridiagonal reduction subroutine DSYTRD [35, 37].

The ratio of level 3 BLAS operations in cases $p_b < b$ and $p_b = b$ exceeds 50% when $n \ge 4b$, while the ratio of level 3 BLAS operations in case $p_b > b$ is always about 50% and does not change with matrix size. □

Corollary 2.2 If $p_b \le b$ where p_b is the algorithmic panel width and b is the block size in the orthogonal block tridiagonal reduction algorithm, the ratio of level 3 BLAS operations increases with matrix size n.

Proof. From Equations 2.15: $ratio_{BLAS\,3} \ge 1 - \frac{3b}{2n}$ and Equation 2.16: $ratio_{BLAS\,3} \ge 1 - \frac{15b}{8n}$, ratio of level 3 BLAS operation increases with n. □

Since the ratio of level 3 BLAS operations is higher in cases 1) and 2) than in case 3) for most reasonable cases ($n \geq 4b$), block tridiagonal reduction using QR factorization should be implemented with $p_b \leq b$. Figure 2.24 shows that the ratio of level 3 BLAS operations exceeds 90% quickly as the matrix size increases.

2.3.4 Back transformation

After the eigenvalues and eigenvectors of $M = V \Lambda V^T$ have been computed, V will be back transformed to the eigenvector matrix of A in a backward order:

$$X = \left(I - W_1 Y_1^T\right)\left(I - W_2 Y_2^T\right)\cdots\left(I - W_{k-1} Y_{k-1}^T\right)\left(I - W_k Y_k^T\right)V$$

where X is the eigenvector matrix of A and $I - W_i Y_i^T$, $1 \leq i \leq k$ is the product of p_b Householder transformations.

Since only Y_i is stored, redundant computation is required to re-construct W_i before p_b Householder transformations can be applied to V by matrix multiplications. The

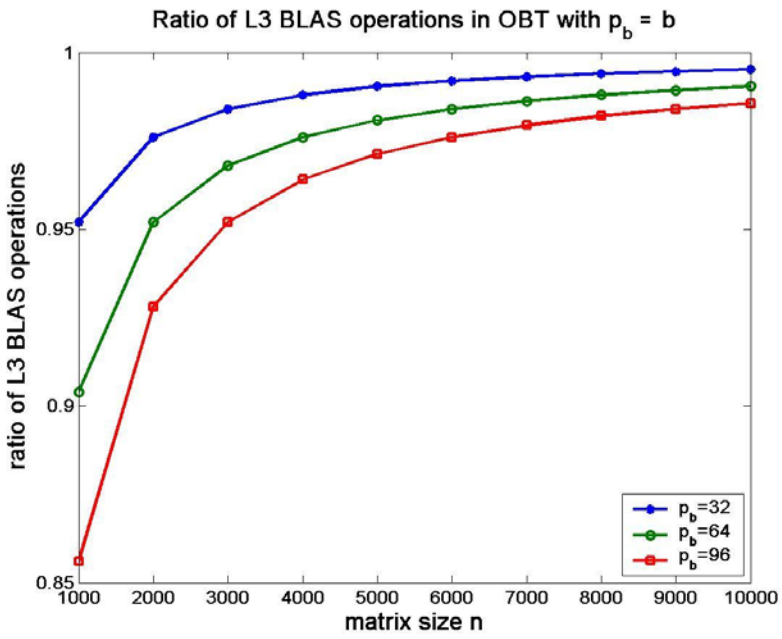

Figure 2.24 Ratio of level 3 BLAS operations in OBR with $p_b = b$.

overhead of re-constructing W_i is $2n_i^2 m_i - \frac{2}{3}n_i^3$ [49] and the computational complexity for applying $I - W_i Y_i^T$ to V is approximately $(4n_i + 1)m_i n$. Here m_i, n_i, p_b are the same as defined in Section 2.3.2 and $q_b = n/p_b$. Total floating-point operations for the back transformation is therefore

$$
\begin{aligned}
flops_{back} &= \sum_{i=1}^{q_b - 1} 2n_i^2 m_i - \frac{2}{3}n_i^3 + (4n_i + 1)m_i n \\
&= \sum_{i=1}^{q_b - 1} 2p_b^2 (n - ip_b) - \frac{2}{3}p_b^3 + (4p_b + 1)(n - ip_b)n \\
&= \sum_{i=1}^{q_b - 1} 2p_b^3 i - \frac{2}{3}p_b^3 + (4p_b + 1)p_b n i \\
&\approx 2n^3 - n^2\left(p_b - \frac{q_b}{2} + \frac{1}{2}\right) + O(np_b^2).
\end{aligned}
$$

3 Parallel block tridiagonal divide-and-conquer (PBD&C) implementation

The BD&C algorithm has the potential of parallelism in that it is divide-and-conquer and recursive in nature. However, the size of each subproblem and the amount of work for solving each of those subproblems at the same level of the recursion is usually different, which leads to workload imbalance. Earlier effort on the parallelization of BD&C achieved modest speedup with 4 – 16 processors [25]. In that implementation, the high performance of the parallel matrix multiplication subroutine in PBLAS [19] cannot be fully exploited due to its storage scheme of matrix sub-blocks. A fine-grained PBD&C is designed here to achieve workload balance and data balance at the same time. Some major issues in such an implementation are: 1) overhead of data communication; 2) order of merging sequence; 3) handling of deflation. We discuss them in detail in the context of PBD&C implementation and give estimation of complexities to help us understand the behavior of PBD&C.

Recall that γ denotes the time for one floating-point operation, α denotes the latency for one communication, β denotes the time to transfer one double precision number, and n_b denotes the block size of the 2D block cyclic parallel matrix distribution. We define LCM as the least common multiple and GCD as the greatest common divisor. For two integers a and b, $LCM(a,b)GCD(a,b) = ab$. The union of two processor grids is called a supergrid. In the computational complexity analyses in this section, we assume 0% deflation unless otherwise specified.

3.1 Data parallelism versus task parallelism

There are different ways to distribute a matrix on a processor grid. Data parallelism distributes data evenly to all the processors and invokes all relevant processors to work on the same task as the algorithm proceeds. Task parallelism assigns each processor to a different task in the algorithm working simultaneously whenever possible. For example, assume we have a block tridiagonal matrix $M \in \mathbb{R}^{n\times n}$ with 4 blocks of equal size $b = n/4$,

as shown in Figure 3.1 and a 4-processor grid $\mathcal{G}$ in the shape of $\begin{bmatrix} p_0 & p_1 \\ p_2 & p_3 \end{bmatrix}$. Each matrix sub-block is highlighted by distinctive color.

We first distribute M using the 2D block cyclic data distribution and assume $n_b = b/2$. The distribution of M as shown in Figure 3.2 is an example of data-parallelism. (Note that due to symmetry, only the lower triangular part is shown as distributed.)

For task parallelism, one could distribute the matrix blocks to the processors as illustrated in Figure 3.3.

One of the advantages of data parallelism is the data distribution and workload balance. However, it has the potential of increased communication resulting in degraded performance. In addition, in problem subdivision, subproblem solution, and at the beginning of the BD&C recursive merges, with data parallel implementation, not all the processors may be working on a single matrix block due to small block size relative to grid size, which causes workload imbalance and a waste of resources. On the other hand, with task parallelism, subsets of processors work on different subproblems independently with reduced communication overhead. However, it may lose data balance and limit the

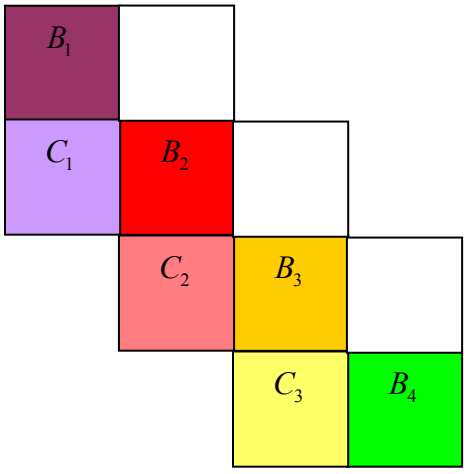

Figure 3.1 A symmetric block tridiagonal matrix with 4 blocks of equal size.

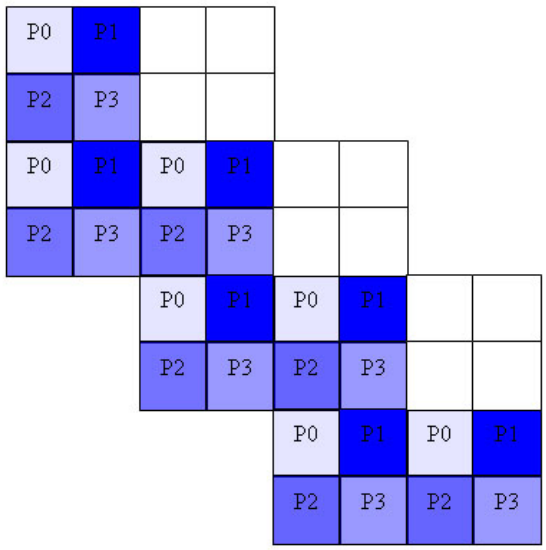

Figure 3.2 Matrix *M* distributed for data parallelism.

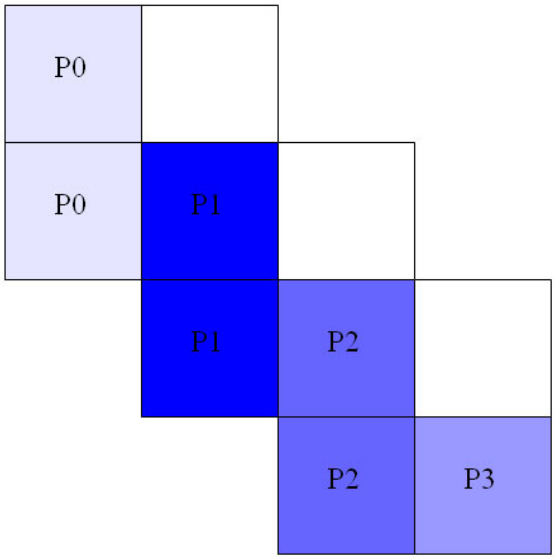

Figure 3.3 Matrix *M* distributed for task parallelism.

size of subproblems to be solved. Moreover, with task parallelism, the overhead of data redistribution before a merging operation in the BD&C algorithm may be large, compared to the computational effort.

To achieve both workload balance and data balance, PBD&C implementation uses a mixed (data/task) parallelism [18]. To be specific, each subproblem B_i will be assigned a group of processors $\mathcal{G}_i$ based upon its anticipated computational complexity, and each group of processors works on a subproblem simultaneously, as shown in Figures 3.4 – 3.6.

Implementation of PBD&C using mixed parallelism involves periodic redistribution of matrix sub-blocks from one processor grid to another (see Sections 3.2.2 and 3.4.1 for details). In what follows, we examine the general data redistribution pattern and communication complexity.

Assume we are given two 1D processor grids: $\mathcal{G}_1 = [p_0\ p_1\ p_2\ p_3]$ and $\mathcal{G}_2 = [p_4\ p_5\ p_6\ p_7\ p_8\ p_9]$. Grid $\mathcal{G}_1$ has 4 processors and $\mathcal{G}_2$ has 6 processors. A matrix $B \in \mathbb{R}^{m\times n}$ is distributed in 1D block cyclic pattern with 12 blocks. Figure 3.7 shows the distribution of B on $\mathcal{G}_1$ and Figure 3.8 shows the distribution of B on $\mathcal{G}_2$. If we redistribute B from $\mathcal{G}_1$ to $\mathcal{G}_2$, then each processor in $\mathcal{G}_1$ sends out three blocks to processors in $\mathcal{G}_2$, and each processor in $\mathcal{G}_2$ receives two blocks from processors in $\mathcal{G}_1$. For example, p_0 sends one block to each of p_4, p_8 and p_6, while p_4 receives one block from each of p_0 and p_2. In general, For each processor in $\mathcal{G}_1$, the number of blocks it sends equals $\frac{LCM(c_1,c_2)}{c_1}$ where c_1 and c_2 are the number of processors in grids $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively. The size of data sent by each processor in $\mathcal{G}_1$ equals mn/c_1. Similarly, for each processor in $\mathcal{G}_2$, the number of blocks it receives equals $\frac{LCM(c_1,c_2)}{c_2}$. The size of data received by each processor in $\mathcal{G}_2$ equals mn/c_2. If a processor needs to send more

$$\begin{bmatrix} B_1 & C_1^T & & \\ C_1 & B_2 & \ddots & \\ & \ddots & \ddots & C_{q-1}^T \\ & & C_{q-1} & B_q \end{bmatrix}$$

Figure 3.4 Block tridiagonal matrix with q diagonal blocks.

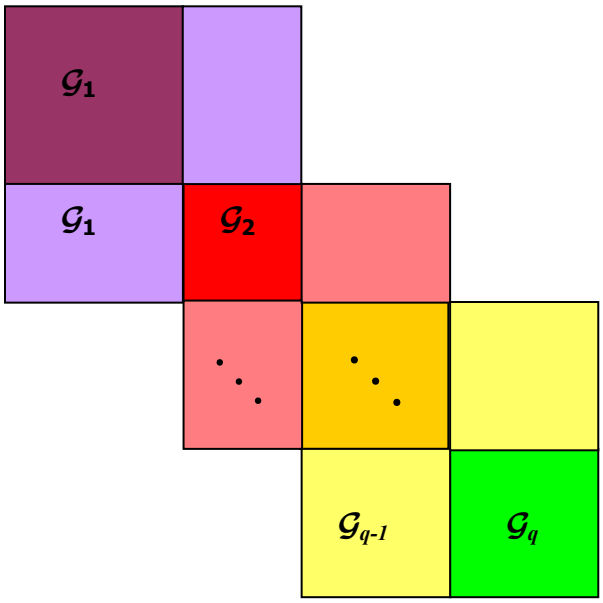

Figure 3.5 Each diagonal block B_i is assigned processor subgrid $\mathcal{G}_i$.

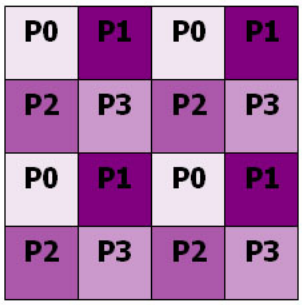

Figure 3.6 Data distribution of block B_1 on a 2×2 processor subgrid $\mathcal{G}_1$.

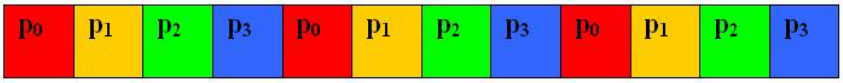

Figure 3.7 Matrix B distributed on 1D grid $\mathcal{G}_1$.

Figure 3.8 Matrix B distributed on 1D grid $\mathcal{G}_2$.

than one block to another processor, those blocks should be packed and sent as one data package.

Analogously, with 2D processor grids, if we redistribute a matrix that is originally distributed in 2D block cyclic pattern on grid $\mathcal{G}_1$ with r_1 processor rows and c_1 processor columns onto grid $\mathcal{G}_2$ with r_2 processor rows and c_2 processor columns, each processor in $\mathcal{G}_1$ sends $\frac{LCM(r_1,r_2)}{r_1} \cdot \frac{LCM(c_1,c_2)}{c_1}$ data packages to processors in $\mathcal{G}_2$, and each processor in $\mathcal{G}_2$ receives $\frac{LCM(r_1,r_2)}{r_2} \cdot \frac{LCM(c_1,c_2)}{c_2}$ data packages from processors in $\mathcal{G}_1$. If grids $\mathcal{G}_1$ and $\mathcal{G}_2$ are disjoint, then each processor in $\mathcal{G}_1$ may be able to send out packages simultaneously without conflict. Otherwise, ordering of send/receive may be required to avoid deadlock because two processors may send to and receive from each other at the same time.

3.2 Parallel subdivision

Suppose we have a block tridiagonal matrix $M \in \mathbb{R}^{n \times n}$ with q number of blocks as shown in Figure 3.4 and p number of processors. Denote size of the *i*-th diagonal block B_i by b_i where $\sum_{i=1}^{q} b_i = n$ and the number of processors assigned to B_i by p_i with $\sum_{i=1}^{q} p_i = p$

where p is the total number of processors available. The *i*-th off-diagonal block C_i has size $b_{i+1} \times b_i$ and approximate rank ρ_i. Processors p_i form a subgrid $\mathcal{G}_i = r_i \times c_i$ where r_i and c_i are the number of rows and columns of the processor subgrid $\mathcal{G}_i$, respectively.

3.2.1 Assign processors to submatrices

The number of processors p_i in the *i*-th subgrid $\mathcal{G}_i$ is determined by

$$p_i = \frac{b_i^3}{\sum_{i=1}^{q} b_i^3} p \tag{3.1}$$

based on the fact that the computational complexity for solving each subproblem is $O\left(b_i^3\right)$. Also, as shown in Figures 3.4 and 3.5, we use the p_i processors assigned to B_i for $1 \le i \le q-1$ to compute the approximate rank of the off-diagonal blocks C_i using the singular value decomposition $C_i = U_i \Sigma_i V_i^T$. Processors p_q assigned to subgrid $\mathcal{G}_q$ will be idle during the SVD computation, but the time for computing the SVD is negligible in comparison to the total time of the PBD&C algorithm, so that it would not lead to noticeable effect on the workload balance.

3.2.2 Distribute a matrix sub-block from one subgrid to another subgrid

To modify the diagonal block: $\tilde{B}_{i+1} = B_{i+1} - U_i \Sigma_i U_i^T - V_{i+1} \Sigma_{i+1} V_{i+1}^T$, left singular vector matrix U_i on processor subgrid $\mathcal{G}_i$ must be redistributed to processor subgrid $\mathcal{G}_{i+1}$ as illustrated in Figure 3.9.

The time for sending U_i to a new processor grid $\mathcal{G}_{i+1}$ is given by

$$t_i^{send} = \alpha \frac{LCM\left(r_i, r_{i+1}\right) LCM\left(c_i, c_{i+1}\right)}{p_i} + \beta \frac{b_{i+1} \rho_i}{p_i}.$$

Note that each processor subgrid except the first one and the last one sends out its copy of U_i and receives a copy of U_{i-1} from its neighbor. The time for processor subgrid $\mathcal{G}_i$ to

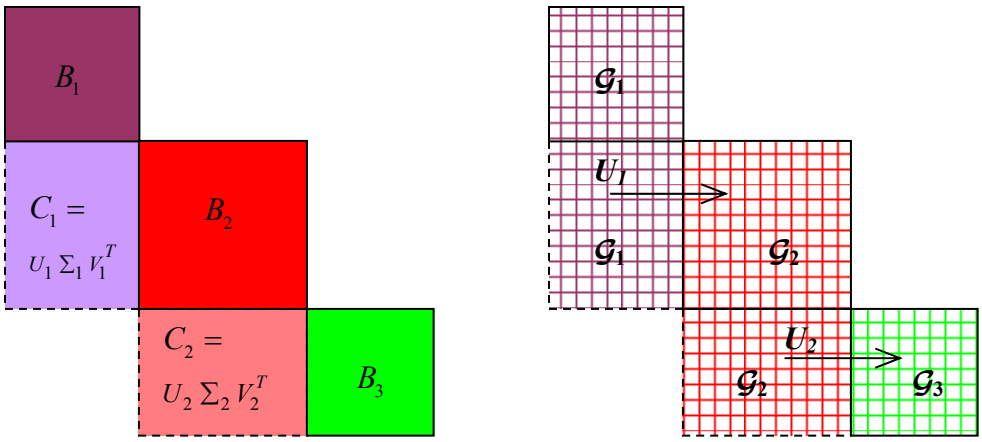

Figure 3.9 Distribute a matrix block from one grid to another.

receive a copy of U_{i-1} from subgrid $\mathcal{G}_{i-1}$ is given by

$$t_i^{recv} = \alpha \frac{LCM\left(r_{i-1}, r_i\right) LCM\left(c_{i-1}, c_i\right)}{p_i} + \beta \frac{b_i \rho_{i-1}}{p_i}.$$

Modern interconnection technology can support overlap of point-to-point communication, which to some extent enables simultaneous sends and receives on a processor grid. In the worst case, when send and receive on one processor subgrid cannot be overlapped, the total time for redistribution of singular vectors equals

$$\begin{aligned} t_i^{redistr1} &= t_i^{send} + t_i^{recv} \\ &= \alpha \left(\frac{LCM\left(r_i, r_{i+1}\right) LCM\left(c_i, c_{i+1}\right)}{p_i} + \frac{LCM\left(r_{i-1}, r_i\right) LCM\left(c_{i-1}, c_i\right)}{p_i} \right) \\ &\quad + \beta \left(\frac{b_{i+1} \rho_i}{p_i} + \frac{b_i \rho_{i-1}}{p_i} \right). \end{aligned} \tag{3.2}$$

All the processor subgrids perform their own data communication with their neighbors simultaneously. Therefore, the total time for singular vector matrix redistribution is $\max_{i=2,q-1} t_i^{redistr1}$ in the worst case. This type of redistribution is required only once in PBD&C.

3.3 Parallel solution of subproblems

Each subgrid can perform the eigen-decomposition of each subproblem independently. There is no communication between any two subgrids; any communication required occurs only within a subgrid. During the solution of subproblems step, all processors are busy solving the subproblem $\tilde{B}_i$ assigned to their subgrid.

3.4 Parallel synthesis of solutions

Parallel synthesis is the most time consuming step of the PBD&C implementation, as is the synthesis step of the sequential BD&C algorithm. Major issues are: (1) before each merging step, submatrices on two subgrids need to be redistributed to its supergrid; (2) during the accumulation of eigenvectors, deflation needs to be handled in a way to minimize communication; and (3) a merging sequence needs to be determined that minimizes both the computational complexity and processor idle time. The dominant term in the complexity of PBD&C is determined by the complexity of the last several steps of the synthesis, as in the sequential BD&C.

The synthesis step of PBD&C may be represented as a binary merging tree of merging operations illustrated in Figure 3.10. The bottom of the merging tree is labeled as merge level 0, and the leaves are the eigensolutions of the modified subproblems $\tilde{B}_1, \tilde{B}_2, \cdots, \tilde{B}_q$, each of size b_i distributed on subgrid $\mathcal{G}_i$ with p_i processors for $1 \le i \le q$. Each pair of eigensolutions is merged simultaneously. Before each merging operation, two subgrids

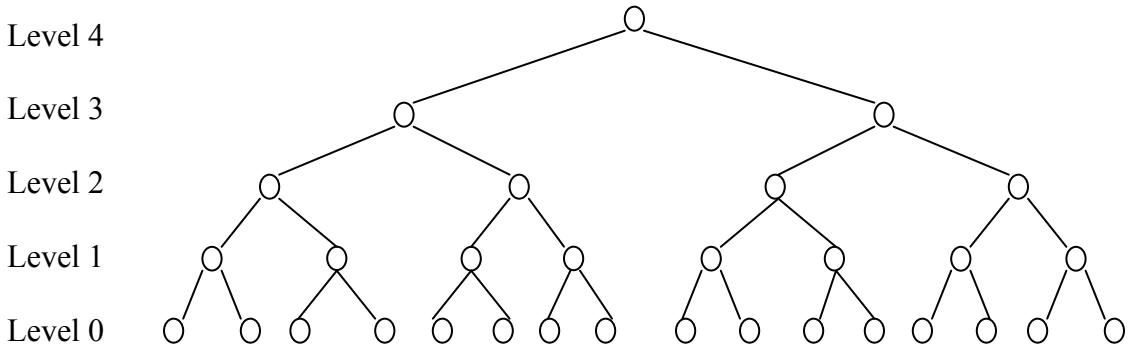

Figure 3.10 Merging tree and level of merging.

that hold the two eigensolutions to be merged need to be combined to form a supergrid, and eigensolutions need to be redistributed as well. The union of two disjoint subgrids $\mathcal{G}_i$ and $\mathcal{G}_{i+1}$ is a supergrid $\mathcal{G} = \mathcal{G}_i \cup \mathcal{G}_{i+1} = r \times c$. This supergrid will hold the eigensolutions of the merged subproblems. After the adjacent merges at the bottom of the merging tree finish, the next level of merge starts until the root of the tree is reached. The height of a node is the longest path from that node to each leaf, and the height of a merging tree is the longest path from the root to the furthest leaf. The root of the merging tree, which is the final merging operation, is labeled as level h. For example, $h = 4$ for Figure 3.10. The merging levels before the final merging are labeled as $h-1$, $h-2$, $\cdots$, 1.

3.4.1 Redistribution of data from two subgrids to a supergrid

Before we start a merging operation, that is, a sequence of matrix multiplications, the two subgrids that hold the two submatrices of eigenvectors must be grouped together to form a supergrid. The corresponding submatrices must be redistributed to the supergrid correspondingly. This type of data redistribution is invoked on each level of the merging tree as the merging operations go up the tree. Figures 3.11 and 3.12 illustrate the redistribution of two submatrices, one from a 2×2 grid and the second from a 2×4 grid, to a 3×4 grid.

In practice, boundaries of submatrices seldom match the natural boundaries of 2D block cyclic distribution. The starting point of a submatrix B_i in the supergrid is not always a multiple of n_b. In such a case, an offset between the two different types of boundaries must be computed for the correct indexing of submatrices in the supergrid.

Assume processors redistribute their data in a canonical order without pipelining. That is, with k processors numbered from 0 to $k-1$, processor 0 sends out its data to processors $1, 2, \ldots, k-1$, then processor 1 sends out its data to processors $0, 2, \ldots, k-1$, and so on, and finally processor $k-1$ sends out its data to processors $0, 1, \ldots, k-2$. Since the time for data redistribution for each processor is

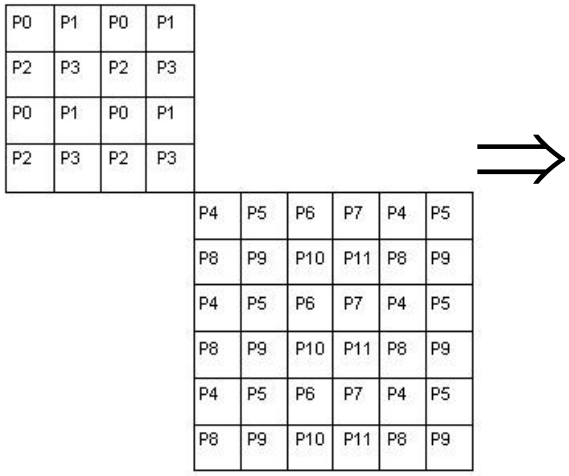

Figure 3.11 The first submatrix held by a 2×2 grid, the second submatrix held by a 2×4 grid.

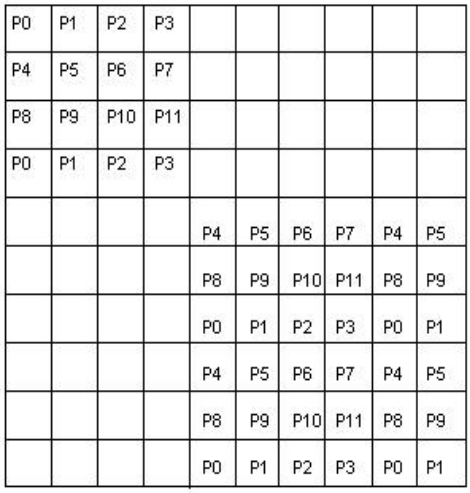

Figure 3.12 Two submatrices redistributed to a 3×4 supergrid.

$$\alpha\left[\frac{LCM(r_i,r)LCM(c_i,c)}{p_i}+\frac{LCM(r_{i+1},r)LCM(c_{i+1},c)}{p_{i+1}}\right]+\beta\left(\frac{b_i^2}{p_i}+\frac{b_{i+1}^2}{p_{i+1}}\right),$$

redistribution of two matrix blocks from two disjoint subgrids to the union of those two subgrids in such an order takes the total time

$$t_i^{redistr2}=\alpha\left[p_i\frac{LCM(r_i,r)LCM(c_i,c)}{p_i}+p_{i+1}\frac{LCM(r_{i+1},r)LCM(c_{i+1},c)}{p_{i+1}}\right]+\beta\left(p_i\frac{b_i^2}{p_i}+p_{i+1}\frac{b_{i+1}^2}{p_{i+1}}\right)$$

$$=\alpha p\left[\frac{p_i}{GCD(r_i,r)GCD(c_i,c)}+\frac{p_{i+1}}{GCD(r_{i+1},r)GCD(c_{i+1},c)}\right]+\beta\left(b_i^2+b_{i+1}^2\right). \qquad (3.3)$$

As one may observe, the communication cost depends on the computer and network specification as well as the shapes of subgrids and supergrids. When the number of rows and columns of a supergrid and its corresponding subgrids are mutually prime, there are p^2 communications to accomplish the data transfer, and the accumulative start up time for communications is high. However, this is typically not the case. In the best case, the frequency of communications can be reduced to $2p$ if two subgrids have the same number of processors.

3.4.2 Merging sequence

Each merging operation of two subproblems includes steps such as solving the secular equation, deflation and accumulation of the eigenvector matrix. Among those steps, the accumulation of eigenvector matrices is by far the most time consuming part. We approximate the computational and communication costs of a merging operation by the matrix multiplications involved.

In the sequential implementation of BD&C, merging starts from off-diagonal blocks with the highest rank, leaving the off-diagonal block with the lowest rank for the final merging operation to reduce the computational complexity of BD&C. This merging sequence is sequential in nature and not completely appropriate for a parallel implementation. For example, consider a block tridiagonal matrix M with p processors

and 4 diagonal blocks of equal sizes, i.e., $q = 4$, $p_1 = p_2 = p_3 = p_4 = \frac{p}{4}$

and $b_1 = b_2 = b_3 = b_4 = \frac{n}{4}$ as shown in Figure 3.13. Further assume $\rho_1 < \rho_2 = \rho_3 = \rho_4$, i.e., the first off-diagonal block has the lowest rank.

If we choose the off-diagonal block with the lowest rank for the final merging operation, the processors in $\mathcal{G}_1$ and $\mathcal{G}_2$ stay idle while processors in $\mathcal{G}_3$ and $\mathcal{G}_4$ handle the merge for blocks B_3 and B_4; then $\mathcal{G}_1$ stays idle while processors in $\mathcal{G}_2$, $\mathcal{G}_3$ and $\mathcal{G}_4$ handle the merge for blocks B_2, B_3 and B_4. If one assumes 0% deflation and neglects the overhead of communication, the total computational time per processor can be approximated by

$$
\begin{aligned}
t_{low\_rank} &= t_{merge(B_3,B_4)} + t_{merge(B_2,B_3,B_4)} + t_{merge(B_1,B_2,B_3,B_4)} \\
&= 2\gamma \left[\frac{(n/2)^3}{p/2}\rho_2 + \frac{(3n/4)^3}{3p/4}\rho_2 + \frac{n^3}{p}\rho_1 \right] \\
&= 2\gamma \left(\frac{13n^3}{16p}\rho_2 + \frac{n^3}{p}\rho_1 \right). \qquad (3.4)
\end{aligned}
$$

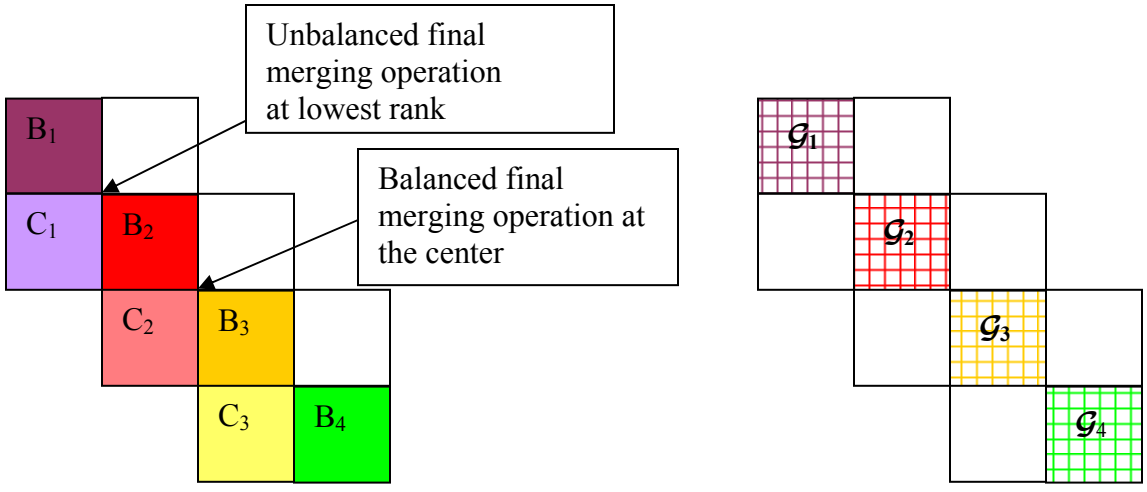

Figure 3.13 A block tridiagonal matrix with 4 blocks of same block size.

Suppose we now neglect the ranks and choose the off-diagonal block C_2 for the final merging operation and obtaining the most balanced merging sequence, then B_1 and B_2 are merged simultaneously to the merging of B_3 and B_4. In this situation, the total computational time per processor of all the merging operations can be approximated by

$$\begin{aligned} t_{balance} &= t_{merge(B_3,B_4)} + t_{merge(B_1,B_2,B_3,B_4)} \\ &= 2\gamma\left[\frac{(n/2)^3}{p/2}\rho_2 + \frac{n^3}{p}\rho_2\right] \\ &= 2\gamma\left(\frac{5n^3}{4p}\rho_2\right). \end{aligned} \tag{3.5}$$

Comparing Equations 3.4 and 3.5, one concludes $t_{low\_rank} < t_{balance}$ only when $\rho_1 < \frac{7}{16}\rho_2$. This indicates that unless the lowest rank of off-diagonal blocks is less than half the rank of the off-diagonal block in the middle of M, choosing a balanced final merging operation keeps all processors busy and achieves a better workload balance, which subsequently leads to less idle time and consequently less total execution time.

Based on the above observation, we determine the position of the off-diagonal block for the final merging operation according to both computational complexity and workload balance.

The merging tree shown in Figure 3.10 has the same number of subproblems on the left and right sides of the final merging operation. In general, this is not the case. In a block tridiagonal matrix M, the off-diagonal block for the most balanced final merging operation is the one closest to the middle of M. At the bottom of the merging tree, subproblems usually have different sizes. As the merging moves up the tree, subproblems on each level continue to have different sizes. In general, the number of matrix sub-blocks on the left side of the final merging operation is different from that on the right side, even if the final merging operation is a balanced one. However, in a balanced final merging operation, it is guaranteed that the amount of workload per processor on both sides of the final merging operation is approximately the same because processors are assigned to subproblems based on problem size.

Figure 3.14 shows a merging tree with different number of subproblems on the left and right sides of the final merging operation. In order to evaluate the computational cost of different merging sequences, one needs to consider not only the rank of the off-diagonal block for the final merging operation, but also the sizes of the subproblems as well as the number of idle processors and their idle time.

Suppose we have a merging tree as shown in Figure 3.14 for a block tridiagonal matrix $M \in \mathbb{R}^{n\times n}$ with q blocks and p processors. Without loss of generality, we assume $p \geq q$ so that each subproblem at the bottom of the merging tree is assigned at least one processor. Otherwise, we may always re-block M to satisfy this assumption.

Let f be the position of the off-diagonal block for the final merging operation and ρ_f be the approximate rank of the corresponding off-diagonal block C_f. The matrix sub-blocks indexed from 1 to f construct a left subtree, while the matrix sub-blocks indexed from $f+1$ to q construct a right subtree. If the height of the left and right subtrees are h_{left} and h_{right}, respectively, then the height of the whole merging tree is $h = \max\left(h_{left}, h_{right}\right)+1$. For example, $h_{left} = 2$, $h_{right} = 3$ and $h = 4$ in Figure 3.14.

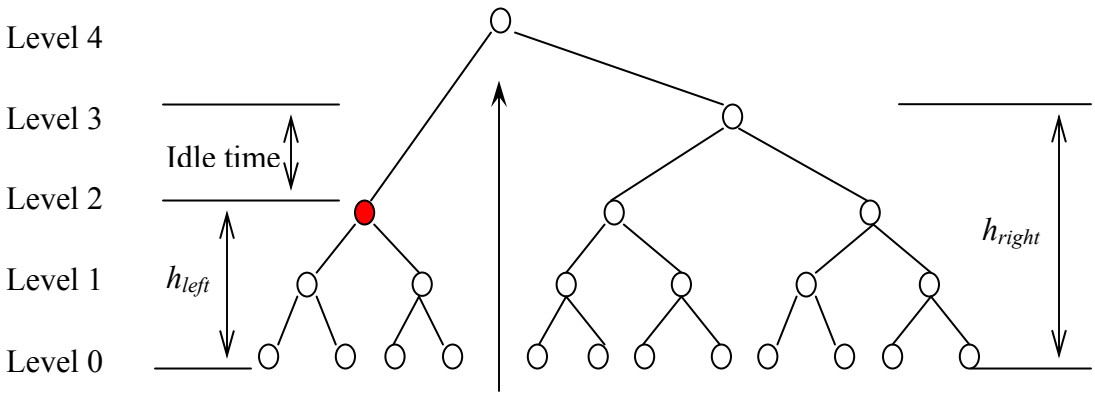

Figure 3.14 Merging tree with different number of subproblems on the left and right of the final merging operation.

We assume that a lower subtree finishes merging before a higher subtree does. The merging process starts from level 0, and loses balance at level $h'' = \min\left(h_{left}, h_{right}\right)+1$. At this stage the processors in the lower subtree stay idle while the processors in the higher subtree keep working. After the higher subtree finishes its merging operations at level $h' = \max\left(h_{left}, h_{right}\right)$, all the processors work together again for the final merging operation. Therefore, the time for all the merging operations is the time to merge the higher subtree plus the time for the final merging operation.

Note that the computational time for the merge of each two adjacent blocks at the lowest level is approximately $2\gamma\rho\left(\sum_{i=1}^{2} b_i\right)^3$, where ρ is the rank of the off-diagonal block that connects the two diagonal blocks to be merged. As the merges proceed up the tree, each block itself may be the result of a previous merge of smaller blocks. The approximate computational time to merge the higher subtree is the sum of the approximate merging time for each level of that subtree:

$$t_1 = 2\gamma \sum_{i=1}^{h'_f} \max_{j=1,\cdots,K_1^f} \left\lceil \frac{\left(\sum_{k=0}^{2^i-1} b_{K_2^f+K_3^f+k}\right)^3 \rho_{K_4^f}}{\sum_{k=0}^{2^i-1} p_{K_2^f+K_3^f+k}} \right\rceil \tag{3.6}$$

where $h'_f = \max\left(h_{left}, h_{right}\right) = h-1$ is the height of the higher subtree,

$$K_1^f = \begin{cases} \left\lceil \frac{q-f}{2^i} \right\rceil - \mathrm{mod}\left(\left\lceil \frac{q-f}{2^{i-1}} \right\rceil, 2\right) & if\ h'_f = h_{right} \\ \left\lceil \frac{f}{2^i} \right\rceil - \mathrm{mod}\left(\left\lceil \frac{f}{2^{i-1}} \right\rceil, 2\right) & if\ h'_f = h_{left} \end{cases}, \tag{3.7}$$

$$K_2^f = (j-1)2^i + 1\ , \tag{3.8}$$

$$K_3^f = \begin{cases} f & if\ h'_f = h_{right} \\ 0 & if\ h'_f = h_{left} \end{cases}, \tag{3.9}$$

$$K_4^f = j2^i - 2^{i-1} + K_3^f\ , \tag{3.10}$$

$$k \le \begin{cases} q - K_2^f - K_3^f & if\ h' = h_{right} \\ f - K_2^f - K_3^f & if\ h' = h_{left} \end{cases}.$$

The computational time for the final merging operation is approximately

$$t_2 = 2\gamma \frac{n^3 \rho_f}{p}. \tag{3.11}$$

Equations 3.6 and 3.11 can be used to compare the computational time of a balanced merge and an unbalanced one with lower rank. Assume the off-diagonal block for a balanced final merging operation is located at position m and the rank of C_m is ρ_m. Correspondingly, assume the off-diagonal block of the final merging operation with the lowest rank is at position l and the rank of C_l is ρ_l.

Theorem 3.1 An unbalanced final merging operation with the lowest rank has less computational time per processor than a balanced final merging operation with higher rank only when

$$\rho_m - \rho_l > \frac{p}{n^3}\left\{\sum_{i=1}^{h_l-1} \max_{j=1,\cdots,K_1^l}\left[\frac{\left(\sum_{k=0}^{2^i-1} b_{K_2^l+K_3^l+k}\right)^3 \rho_{K_4^l}}{\sum_{k=0}^{2^i-1} p_{K_2^l+K_3^l+k}}\right] - \sum_{i=1}^{h_m-1} \max_{j=1,\cdots,K_1^m}\left[\frac{\left(\sum_{k=0}^{2^i-1} b_{K_2^m+K_3^m+k}\right)^3 \rho_{K_4^m}}{\sum_{k=0}^{2^i-1} p_{K_2^m+K_3^m+k}}\right]\right\} \tag{3.12}$$

where h_l is the height of the merge tree with the lowest rank for the final merging operation, h_m is the height of the merge tree with balanced final merging operation, and K_1^l, K_2^l, K_3^l, K_4^l, K_1^m, K_2^m, K_3^m and K_4^m can be computed by Equations 3.7 – 3.10.

Proof. Computational time per processor for an unbalanced merge tree with the lowest rank for the final merging operation is

$$\begin{aligned} t_{low\_rank} &= t_1^{low\_rank} + t_2^{low\_rank} \\ &= 2\gamma \sum_{i=1}^{h_l'} \max_{j=1,\cdots,K_1^l}\left[\frac{\left(\sum_{k=0}^{2^i-1} b_{K_2^l+K_3^l+k}\right)^3 \rho_{K_4^l}}{\sum_{k=0}^{2^i-1} p_{K_2^l+K_3^l+k}}\right] + 2\gamma \frac{n^3 \rho_l}{p} \end{aligned} \tag{3.13}$$

where $h_l' = \max\left(h_{left}^{low\_rank}, h_{right}^{low\_rank}\right) = h_l - 1$ is the height of the higher subtree of the unbalanced final merging operation,

$$K_1^l = \begin{cases} \left\lceil \frac{q-l}{2^i} \right\rceil - \mathrm{mod}\left(\left\lceil \frac{q-l}{2^{i-1}} \right\rceil, 2 \right) & if\ h_l' = h_{right}^{low\_rank} \\ \left\lceil \frac{l}{2^i} \right\rceil - \mathrm{mod}\left(\left\lceil \frac{l}{2^{i-1}} \right\rceil, 2 \right) & if\ h_l' = h_{left}^{low\_rank} \end{cases},$$

$$K_2^l = \left(j-1\right)2^i + 1\ ,$$

$$K_3^l = \begin{cases} l & if\ h_l' = h_{right}^{low\_rank} \\ 0 & if\ h_l' = h_{left}^{low\_rank} \end{cases},$$

and $K_4^l = j2^i - 2^{i-1} + K_3^l$.

Computational time per processor for a merge tree with balanced final merging operation is

$$\begin{aligned} t_{balance} &= t_1^{balance} + t_2^{balance} \\ &= 2\gamma \sum_{i=1}^{h_m'} \max_{j=1,\cdots,K_1^m} \left\lceil \frac{\left(\sum_{k=0}^{2^i-1} b_{K_2^m + K_3^m + k} \right)^3 \rho_{K_4^m}}{\sum_{k=0}^{2^i-1} p_{K_2^m + K_3^m + k}} \right\rceil + 2\gamma \frac{n^3 \rho_m}{p} \end{aligned} \qquad (3.14)$$

where $h_m' = \max\left(h_{left}^{balance}, h_{right}^{balance}\right) = h_m - 1$ is the height of the higher subtree of the balanced final merging operation,

$$K_1^m = \begin{cases} \left\lceil \frac{q-m}{2^i} \right\rceil - \mathrm{mod}\left(\left\lceil \frac{q-m}{2^{i-1}} \right\rceil, 2 \right) & if\ h_m' = h_{right}^{balance} \\ \left\lceil \frac{m}{2^i} \right\rceil - \mathrm{mod}\left(\left\lceil \frac{m}{2^{i-1}} \right\rceil, 2 \right) & if\ h_m' = h_{left}^{balance} \end{cases},$$

$$K_2^m = \left(j-1\right)2^i + 1\ ,$$

$$K_3^m = \begin{cases} m & if\ h_m' = h_{right}^{balance} \\ 0 & if\ h_m' = h_{left}^{balance} \end{cases},$$

and $K_4^m = j2^i - 2^{i-1} + K_3^m$.

Decision of which off-diagonal block to use for the final merging is based on the difference between Equations 3.13 and 3.14. If the difference between the ranks of the off-diagonal block in the middle and the one with the lowest rank satisfies the following condition:

$$t_{balance} - t_{low\_rank} > 0, \qquad (3.15)$$

then the off-diagonal block with the lowest rank will be used for the final merging operation. Otherwise, the off-diagonal block in the middle will be preferred. Replacing $t_{balance}$ and t_{low_rank} in Inequality 3.15 with Equations 3.13 and 3.14 yields Inequality 3.12 and completes the proof. □

As examples, suppose we are given a symmetric block tridiagonal matrix M of order 2000 with 100 diagonal blocks of equal size 20. Each diagonal block is assigned 1 processor; thus, the total number of processors available is 100. If all the off-diagonal blocks have the same rank, then the final merging operation should be located at off-diagonal block 50 in the middle of M. If only one off-diagonal block has rank of 0, then it should be chosen for the final merging operation no matter where it is located. If all off-diagonal blocks including the one in the middle have full rank 20 and only one off-diagonal block has rank 10, then based on Theorem 3.1, the algorithm should choose the off-diagonal block with rank 10 for the final merging operation if its index is within the range 70 – 130 for minimal execution time.

3.4.3 Deflation

The efficiency of BD&C greatly depends on deflation. With lowered accuracy requirement, the occurrence of deflation is very high and the amount of work in the eigenvector accumulation is significantly reduced.

Consider the eigenvector accumulation stage (see Section 2.1.3). Let Z be a block diagonal eigenvector matrix of subproblems, and V be the eigenvector matrix of $D + yy^T$. Because V is modified to $\tilde{V} = P_{type}P_{deflate}GV$, Z must be modified to $\tilde{Z} = ZG^T P_{deflate}P_{type}$, where G is an orthogonal matrix that accumulates all Givens rotations to deflate eigenvalues in $D + yy^T$, $P_{deflate}$ is the permutation matrix that moves all deflated

eigenvectors of V to the bottom, and P_{type} is the permutation matrix that groups columns of $\hat{Z} = ZP_{deflate}$ into four types. Those four types are: 1) matrix columns with zeros in lower part; 2) matrix columns that are dense; 3) matrix columns with zeros in upper part; and 4) matrix columns that are related to deflated eigenvectors. In BD&C and PBD&C, only $\hat{Z}$ for the first rank-one modification in a merging operation has such a matrix structure (see Figure 2.2). $\hat{Z}$ for the rest of the rank-one modifications has only two types of columns: non-deflated and deflated (see Figure 2.3). The purpose of permuting by P_{type} is to reduce the amount of computation in the matrix multiplication $Q = \tilde{Z}\tilde{V}$.

In a sequential implementation, the cost of matrix permutation is trivial compared to the computational cost. In a parallel implementation, the cost of communication between processors can not be neglected for frequent swaps of matrix columns. Suppose the deflated eigenvectors of $\hat{Z}$ in the second rank-one modification of a merging operation are distributed as shown in Figures 3.15 and 3.16. The 2×2 processor grid in this example has the shape $\begin{bmatrix} p_0 & p_1 \\ p_2 & p_3 \end{bmatrix}$ with $p = 4$, $r = 2$ and $c = 2$. The column blocks with vertical lines represent the matrix columns that are grouped into the deflated type.

If deflated eigenvectors in $\hat{Z}$ are permutated to the right end of the matrix to construct $\tilde{Z}$ as in the sequential algorithm and as shown in Figures 3.17 and 3.18, communication costs are incurred by swapping matrix columns residing on different processor columns. With high deflation rate, frequent column swaps will occur, and the performance will be degraded.

A strategy used in the ScaLAPACK subroutine PDSYEVD for tridiagonal eigenvalue problems is to permute columns of $\hat{Z}$ that reside local on each processor column into four groups shown in Figures 2.2 and 2.3, instead of a global permutation [92]. In that implementation, the deflation counted is the minimum of the deflation on each processor

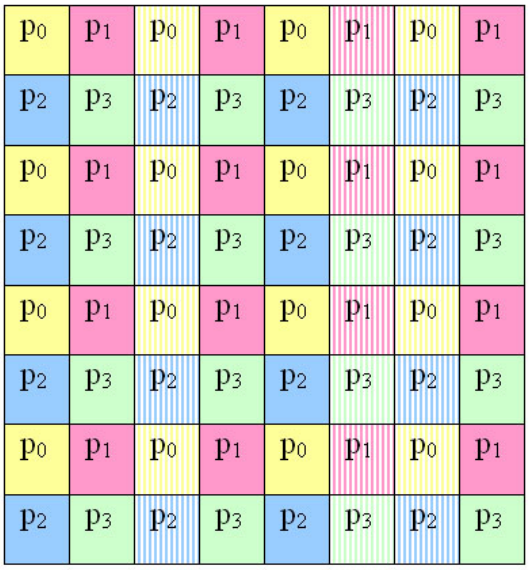

Figure 3.15 Matrix $\hat{Z}$ before grouping – matrix point of view.

Figure 3.16 Matrix $\hat{Z}$ before grouping – processor point of view.

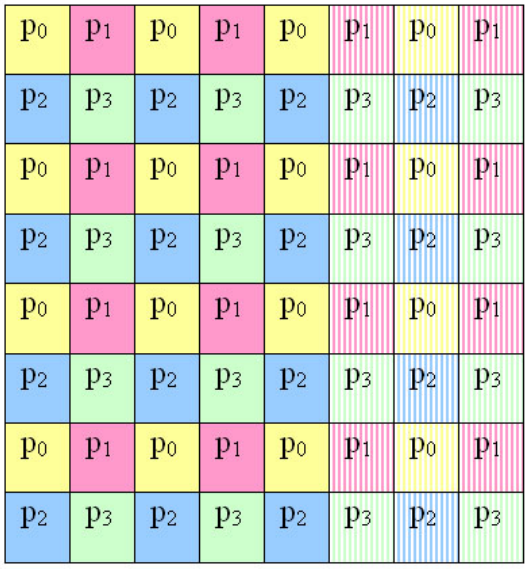

Figure 3.17 Group columns based on their structures – matrix point of view.

Figure 3.18 Group columns based on their structures – processor point of view.

column as illustrated by the column blocks with vertical lines in Figures 3.19 and 3.20. Since processor column $\begin{Bmatrix} p_0 \\ p_2 \end{Bmatrix}$ has two column blocks of deflated eigenvectors and processor column $\begin{Bmatrix} p_1 \\ p_3 \end{Bmatrix}$ has only one column block of deflated eigenvectors, only one column block of deflated eigenvector on $\begin{Bmatrix} p_0 \\ p_2 \end{Bmatrix}$ will be counted, and the column marked with diagonal bars is not counted as deflated eigenvectors. Therefore, the number of deflated eigenvectors incorporated into the algorithm usually does not equal to the number of all deflated eigenvectors, because each processor column typically has a different deflation count. In an example of a most pathetic case, imagine that half of the eigenvectors are deflated, and they are all on processor column $\begin{bmatrix} p_0 \\ p_2 \end{bmatrix}$. No eigenvectors on processor column $\begin{bmatrix} p_1 \\ p_3 \end{bmatrix}$ are deflated. Then the global deflation is zero, not 50%. However, since matrices are distributed in 2D block cyclic pattern, such a case would be extremely rare.

In ScaLAPACK subroutine PDSYEVD, good speedup is obtained although the matrix multiplications performed are not of minimal size [92]. In our test cases of PBD&C as given in Section 5, using the same strategy for matrix re-grouping, an average of 5% less deflation count is observed, which does not significantly degrade the performance of PBD&C.

3.4.4 Complexity of merging

The time complexity of merging operations depends on the matrix structure, i.e., the size of each subproblem to be solved, the approximate ranks of the off-diagonal blocks, the degree of deflation, and the time parameters for floating-point operation and data communication. Those parameters depend both on machine specifications and network

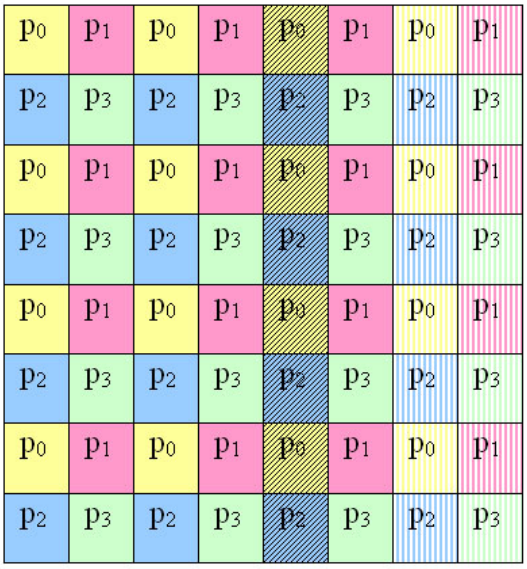

Figure 3.19 Move deflated eigenvectors within processor column – matrix point of view.

Figure 3.20 Move deflated eigenvectors within processor column – processor point of view.

connection that could vary drastically from system to system. In the analysis of computational and communication complexity of merging, we assume no deflation because this parameter varies with matrices to be computed and cannot be predicted before computation starts.

If the sizes of two subproblems to be merged are b_1 and b_2 with $b_1 + b_2 = b$, submatrix 1 has been assigned $p_1 = r_1 \times c_1$ processors, and submatrix 2 has been assigned $p_2 = r_2 \times c_2$ processors with $p_1 + p_2 = p = r \times c$, $r = c = \sqrt{p}$, and the rank of a merging operation is ρ, then the dominant cost of computation and communication per processor without deflation is that of matrix redistribution and parallel matrix multiplications.
Using a ring topology for matrix multiplication, the total cost for one matrix multiplication can be approximated by [94]

$$t_{multiplication} = \frac{2b^3}{p}\gamma + 2\left(b + 2\sqrt{p} - 3\right)\left(\alpha + \frac{b}{\sqrt{p}}\beta\right).$$

Using Equation 3.3, the cost of redistributing matrix blocks is

$$t_{redistribution} = \alpha p\left[\frac{p_1}{GCD(r_1, r)GCD(c_1, c)} + \frac{p_2}{GCD(r_2, r)GCD(c_2, c)}\right] + \beta\left(b_1^2 + b_2^2\right).$$

Total cost for a merging operation includes one matrix redistribution and ρ matrix multiplications:

$$\begin{aligned} t_{one\_merge} &= t_{redistribution} + \rho t_{multiplication} \\ &= \alpha p\left[\frac{p_1}{GCD(r_1, r)GCD(c_1, c)} + \frac{p_2}{GCD(r_2, r)GCD(c_2, c)}\right] + \beta\left(b_1^2 + b_2^2\right) + \\ &\quad \rho\left[\frac{2b^3}{p}\gamma + 2\left(b + 2\sqrt{p} - 3\right)\left(\alpha + \frac{b}{\sqrt{p}}\beta\right)\right]. \end{aligned} \tag{3.11}$$

To simplify the complexity analysis, suppose we have a block tridiagonal matrix M with q diagonal blocks of equal sizes b, each off-diagonal block has same approximate rank ρ, and q is power of 2. The depth of the merging tree is therefore $d = \log_2 q$. Assign s processors to each diagonal block with $s = r \times c$ being a square subgrid. The

total number of processors in use is $p = sq$. Under the above assumptions, the total cost for all the merging operations is given by

$$
\begin{aligned}
t_{all\_merge} &= \sum_{i=1}^{d} t_{one\_merge}^{i} \\
&= \sum_{i=1}^{d} \gamma\left(\frac{2^{2i+1}\rho b^3}{s}\right) + \alpha\left(2^{i+1}s + 2^{i+1}b\rho + 4\sqrt{2^i s}\rho - 6\rho\right) + \\
&\qquad \beta\left(2^{i-1}qb^2 + 2^{i+2}b\rho + \frac{2^{2i+1}b^2 - 6b2^i}{\sqrt{2^i s}}\rho\right) \\
&\approx \frac{8n^3\rho}{3p}\gamma + (4p + 4n\rho)\alpha + \left(n^2 + \frac{3.1n^2}{\sqrt{p}}\rho + 8n\rho\right)\beta + \\
&\qquad O\left(\frac{nb^2\rho}{p}\right)\gamma + O\left(\rho\sqrt{p}\right)\alpha + O\left(\frac{n\rho}{\sqrt{p}} + nb\right)\beta .
\end{aligned}
\tag{3.12}
$$

Since the time for problem subdivision and subproblem solution is trivial compared to the time of subproblem synthesis and merging operations dominate computational and communication complexity of the synthesis step, the total cost of PBD&C can be approximated by the leading terms of the merging cost in Equation 3.12, i.e.,

$$
t_{PBD\&C} \approx \frac{8n^3\rho}{3p}\gamma + (4p + 4n\rho)\alpha + \left(n^2 + \frac{3.1n^2\rho}{\sqrt{p}} + 8n\rho\right)\beta . \tag{3.13}
$$

Among those leading terms, $4p\alpha + n^2\beta$ is the cost for redistributions of all submatrices, and $\frac{8n^3\rho}{3p}\gamma$ is the computational cost, which equals that of the sequential BD&C divided by the number of processors p. Other leading terms in the communication cost are those incurred by data transfer in matrix multiplications.

With a high percentage of deflation, which usually occurs with a lower accuracy requirement for the computed eigenpairs, the cost of computation can be greatly reduced. Equation 3.13 also shows one limitation of the PBD&C: If the ranks of the off-diagonals are high, especially the rank for the last merging operation, the time complexity of

PBD&C increases as a multiple of n^3. Therefore, block tridiagonal matrices with low ranks for off-diagonal blocks are preferred whenever possible.

4 Toward block tridiagonal matrix

Most matrices generated from real application do not have block tridiagonal structure except for the trivial case of $q = 2$. Some of them may have usable structure, and some may not. In either case, pre-processing techniques are necessary to transform matrices into block tridiagonal form. The type of pre-processing techniques used depends on the characteristics of the original matrix, which we will divide into two groups.

The first type includes matrices that are "effectively" sparse, meaning that most of their entries can be neglected without affecting their eigenvalues to the prescribed accuracy. For those matrices, threshold methods and symmetric permutation will be applied to the original matrix in an effort to obtain a suitable block tridiagonal matrix.

The second type includes matrices without properties useful for compressing into blocked form with little computational effort. For these matrices, orthogonal transformations will be applied to reduce the original matrix to a block tridiagonal matrix.

4.1 Parallel block tridiagonalization (PBT) of "effectively" sparse matrix

If an input dense matrix A is "effectively" sparse, the parallel block tridiagonalization (PBT) algorithm will be used to construct a block tridiagonal matrix to approximate A. The differences in eigenvalues of the resultant block tridiagonal matrix and the original matrix are bounded by $\tau \|A\|$, where τ is the prescribed tolerance. In this section, we first discuss the disadvantage of using a 2D block cyclic matrix distribution in PBT, followed by the 6-step PBT implementation using a 1D column block matrix distribution.

4.1.1 1D column block matrix distribution for PBT

The block tridiagonalization (BT) algorithm [4] is heuristic and inherently sequential. The floating-point operations in the algorithm are mainly comparisons and additions, and typically its operation count is $O(n^2)$ [4]. Since the original matrix A is symmetric, an operation on any entry a_{ij} inevitably involves its symmetric counterpart a_{ji}. If the matrix

is not distributed properly, the performance of PBT could degrade severely as the matrix size n and the number of processors p increases.

The 2D block cyclic matrix distribution, which is frequently used in scalable parallel dense matrix algorithms, is not the most suitable data distribution pattern for the task of block tridiagonalization by the BT algorithm. As an example, in the target threshold step of the BT algorithm (see Section 2.2.1, *step 4*), the sum of the absolute values of the elements to be eliminated in each column of matrix A'' is monitored. Matrix elements are traversed diagonally in the order shown in Figure 2.11. If the parallel implementation directly follows this strategy, for each pair of symmetric entries checked, there will be two types of communications: 1) two send/receive between the two processors that hold a_{ij} and a_{ji} so that they can determine whether the symmetric matrix entries can be dropped simultaneously; 2) one broadcast so that all other processors containing elements in those two columns can update the column sums of the error matrix E. For a processor grid with $p = r \times c$ processors where r is the number of rows and c is the number of columns in the processor grid, communication overhead invoked by type 1) is of $O\left(n^2\right)$, and that invoked by type 2) is of $O\left(n^2 \log r\right)$. To reduce the communication overhead, one possible alternative is to implement the sequential algorithm in blocked pattern as shown in Figure 4.1 where matrix A is distributed on a 2×2 processor grid.
The sum of the dropped elements can be checked block by block along the off-diagonals. But even so, the communication cost still sums to $\frac{n^2}{n_b^2}\left(1+\log r\right)\left(\alpha+\beta\right)$, which is a function of $n^2\left(1+\log r\right)$ since n_b is a constant.

As in the BT algorithm, the matrix must be traversed column-wise numerous times in the PBT algorithm. Based on this fact, intuitively, a 1D column block distribution with n/p matrix columns assigned to each processor, as shown in Figure 4.2, for the matrix is most desirable, and will be used for the PBT algorithm.

If the original input matrix A is distributed in a 2D block cyclic pattern, then it must be redistributed from 2D to 1D for the PBT algorithm. If we assume that the system

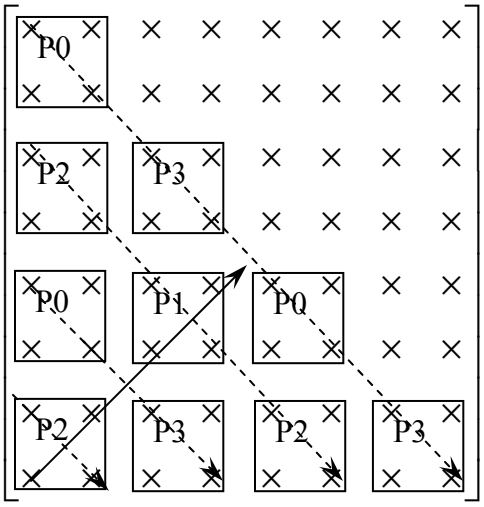

Figure 4.1 Traverse off-diagonals block by block.

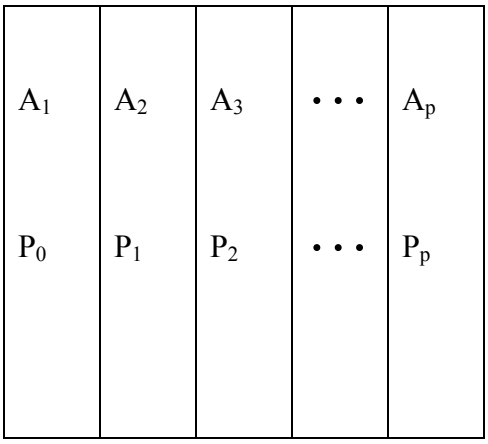

Figure 4.2 Matrix A distributed in column blocks.

buffer is large enough so that each message can be sent and received without being partitioned into several smaller packages and point-to-point communication (i.e., send and receive) cannot be overlapped, then the total communication cost in the worst case for matrix redistribution from 2D block cyclic pattern to 1D column block pattern is

$$t_{2D\to 1D} = p^2\alpha + n^2\beta .$$

After the matrix has been redistributed, each processor holds n/p columns, and the parallel block tridiagonalization is then applied.

4.1.2 The 6-step PBT algorithm

As in the sequential BT algorithm, there are also 6 steps in the PBT implementation, and the accuracy tolerance τ is partitioned as $\tau = \tau_1 + \tau_2$ for target threshold and optional target block reduction, respectively.

Step 1. Parallel global threshold with $\sqrt{\tau}\|A\|$.

This step is an embarrassingly parallel process. Every processor drops all elements $a_{ij} < \sqrt{\tau}\|A\|$, and stores indices of all elements $a_{ij} \geq \sqrt{\tau}\|A\|$ in compressed sparse row (CSR) format. The resultant matrix A' is expected to be very sparse and all its nonzero entries can be stored on one processor. Therefore, after thresholding, each processor sends its vectors of indices of nonzeros to a master processor. The master processor stores indices of all the nonzeros of A'. The collection of indices of nonzeros takes $2p\alpha + (n + nnz_1)\beta$ communication time where nnz_1 is the number of nonzeros in A'. No floating-point operations are involved in this step.

Step 2. Matrix reorder.

The most thoroughly studied and parallelized sparse matrix ordering algorithms are nested dissection and minimum degree algorithms, which are used to minimize the fill-ins during LU factorization of matrices in sparse linear systems [53]. Scalable and

efficient parallel implementations of those algorithms such as ParMetis [60] are available. However, the purpose of matrix ordering in the PBT is to minimize the bandwidth of a sparse matrix, and the nested dissection and minimum degree methods do not directly attack this objective.

Since the matrix after global threshold is expected to be very sparse and can be stored on the local memory of one processor and the reordering consumes a small fraction of computational time of the PBT, we do not parallelize the reordering step. Instead, only the master processor that contains the indices of all nonzeros of A' performs matrix reordering. The Gibbs-Poole-Stockmeyer (GPS) algorithm [23, 46, 64], , which directly attacks the bandwidth minimization problem, is used in the BT algorithm and will be used in the PBT algorithm, while all other processors stay idle. After the permutations are determined, the master processor broadcasts the permutation matrix P to all other processors.

Step 3. Parallel symmetric permutation of A.

The permutation matrix P from *step 2* is used to permute the matrix A to produce the matrix $A'' = P^T AP$. Parallel symmetric matrix permutation can be an expensive step. As shown by the blue arrow in Figure 4.3, if two matrix columns are on different processors, the swap of those two columns invokes communication. In such a case, the communication cost of each swap is $2\alpha + 2n\beta$. In 1D column block distribution, permutation of matrix rows does not involve any communication. If two rows of a matrix are to be swapped, local data on each processor are exchanged as shown by the red arrow in Figure 4.3. Thus, the worst-case communication cost is bounded by $n\alpha + n^2\beta$.

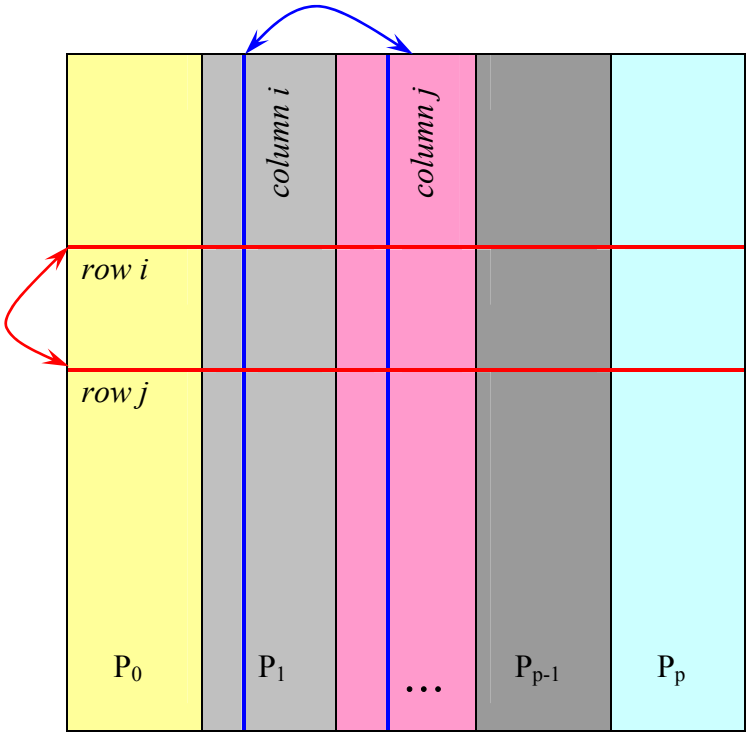

Figure 4.3 Swaps of rows and columns in parallel matrix permutation.

Because of the potentially heavy communication, matrix permutation is executed only when it can significantly reduce the bandwidth of A'. For the PBT algorithm, we permute A when the bandwidth can be reduced by at least 20%.

Step 4. Parallel target threshold with $\tau_1 \|A\|$**.**

In the sequential BT algorithm, all elements far away from the diagonal of matrix A'' are eliminated if their influence on the error of any eigenvalue is less than $\tau_1 \|A\|$. The resultant matrix is $A''' = A'' + E$, with $\|E\|_1 \le \tau_1 \|A\|$. Here one traverses the off-diagonals of A'' while checking the 1-norm of the error matrix E. For each element a''_{ij} in the lower triangular part of matrix A'' that is checked for elimination, its symmetric counter part a''_{ji} in the upper triangular of A'' must also be checked. Elements a''_{ij} and a''_{ji} can be

dropped only when the sum of the absolute values of the dropped elements in both the *i*-th and the *j*-th column of A'' is less than $\tau_1 \|A\|$.

In a parallel matrix distribution, chances are that entries a''_{ij} and a''_{ji} are often on two different processors requiring communications between those two processors, in order to inform each other whether a''_{ij} and a''_{ji} can be dropped simultaneously or not. On the average, this leads to $O(n^2)$ communications.

The communication overhead can be reduced drastically if elements on each processor can be checked independently without communication. For this purpose, the target threshold algorithm is modified. The error bound τ_1 is further split into two equal parts of $\frac{1}{2}\tau_1$, and the error matrix E is also split into two parts: $E = E_1 + E_2$, where E_1 is an upper triangular matrix and E_2 is a lower triangular matrix.

The lower triangular part of A'' is first checked column by column. An element in the lower triangular part of A'' can be eliminated if the sum of the absolute values of the dropped elements in that column is less than $\frac{1}{2}\tau_1 \|A\|$. This guarantees that the error matrix E_1 satisfies $\|E_1\|_1 \le \frac{1}{2}\tau_1 \|A\|$. After that, the sum of the absolute values of all dropped elements in each column of A'' is broadcasted so that each processor contains a copy of the accumulated error for each matrix column. Then the upper triangular part of A'' is checked in a similar way. This guarantees that the error matrix E_2 which contains all the dropped elements in the upper triangular part of A'' satisfies $\|E_1 + E_2\|_1 \le \tau_1 \|A\|$.

For each eliminated element a''_{ij}, its symmetric counter part a''_{ji} is not necessarily eligible for elimination, and vise versa. In general, from the above procedure, E_1 does not equal E_2^T. Therefore, the sum of those two matrices, $E_1 + E_2$, is not symmetric. Since matrices E and A''' must be symmetric, we need to symmetrize $E_1 + E_2$. For the

i-th column and row of A''', $1 \le i \le n$, the row index of the last nonzero of column i and the column index of the last nonzero of row i is compared. The larger index is chosen as the index of the last nonzero for the i-th row and column as shown in Figures 4.4 and 4.5.

The total number of communication in this modified parallel target threshold algorithm is only $(4\alpha + 4n\beta)\log p$.

By using the above approach in our parallel target threshold algorithm, we may not be able to drop as many elements as we mathematically could and as in the sequential BT algorithm. However, the difference in the bandwidths produced by BT and PBT is typically small (less than 10%) as our test results of application matrices show.

Step 5. Covering A'''.

After the parallel target threshold step, all processors obtain the row indices of the last nonzero entries in each column of matrix A'''. Each processor redundantly determines the sizes of the diagonal blocks as in the sequential BT algorithm, so that the resulting block tridiagonal matrix contains all the matrix elements that are effectively nonzero (i.e., nonzeros in A''').

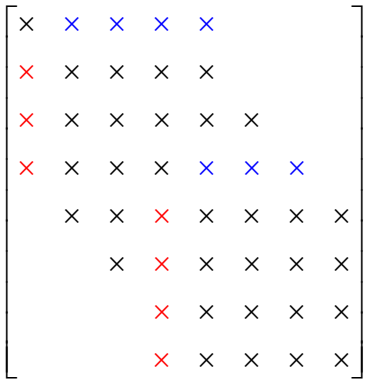

Figure 4.4 A''' after separate lower and upper triangular eliminations.

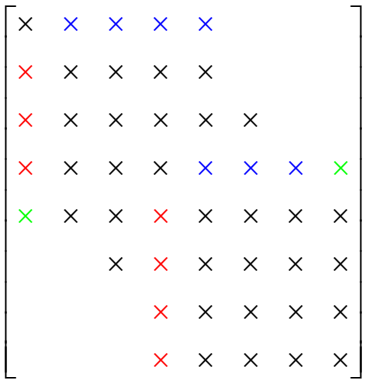

Figure 4.5 Symmetrize A''' by adding back nonzeros.

Step 6. Parallel target block reduction using $\tau_2 \|A\|$**.**

The sequential BT algorithm provides the option of Target Block Reduction (TBR) to produce a few small blocks in matrix A''' for a lower computational complexity in the merging operations of the BD&C algorithm. In a merging operation of BD&C, a lower rank of the off-diagonal block leads to a lower computational complexity. Since the ranks of off-diagonal blocks are not available during block tridiagonalization, we use the smaller dimension of an off-diagonal block as an approximation to its rank. TBR uses sensitivity analysis to check elements in each column/row of an off-diagonal block from outside toward inside for elimination. For the sensitivity analysis, approximations to the eigenvectors are required (see Section 2.2.1 *step 6*). If approximate eigenvectors are not available, we may set $\tau_1 = \tau$ and $\tau_2 = 0$ so that this optional step is not applied.

For the parallel implementation of sensitivity analysis, we assume that the approximate eigenvector matrix is distributed in 2D block cyclic pattern on a processor grid with r processor rows and c processor columns as would typically be the case. When rows of the approximate eigenvector matrix are required, they are sent from several processors to one processor. That is, for each entry a'''_{ij} to be checked, the *i*-th and *j*-th row of the eigenvector matrix need to be sent to the processor that possesses a'''_{ij} (see Equation 2.7), which costs $2c\alpha + 2n\beta$ communication time. When several matrix entries in the same column are checked for elimination, the strategy used in parallel TBR is to send all the relevant rows in the eigenvector matrix to the processor that is applying the sensitivity analysis. For example, as shown in Figures 4.6 and 4.7, if we want to check elements b_{25}, b_{52}, b_{35} and b_{53} of a matrix *B* for elimination, rows 2, 3, and 5 of the eigenvector matrix Z (red shade in Figure 4.7) are sent to P_2, since b_{25} and b_{35} are both on processor P_2. The updated block size will then be broadcast to all other processors.

To find diagonal blocks eligible for sensitivity analysis, parallel TBR starts with the smallest diagonal block. If there are several diagonal blocks with the same size, then the diagonal block closest to the middle of A''' will be selected. The reduction of the size of

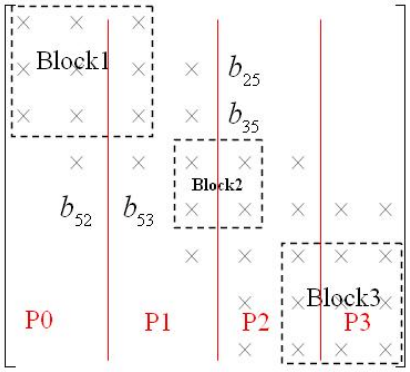

Figure 4.6 Check matrix entries (2,5), (3,5), (5,2) and (5,3).

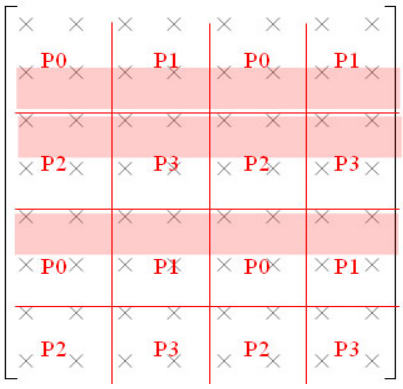

Figure 4.7 Rows in the eigenvector matrix Z for sensitivity analysis.

one diagonal block leads to the expansion of its neighboring diagonal block(s). To avoid oscillation in block sizes, after a diagonal block has been compressed, it should not be expanded any more. Next, the second smallest diagonal block is selected in a similar manner for sensitivity analysis, and so on.

In the sequential BT algorithm, all eligible diagonal blocks are checked for block size reduction. However, this may be too costly for PBT since communication overhead for collecting and distributing rows of the eigenvector matrix can be prohibitive. In out tests, the TBR step usually takes about one-half of the execution time of PBT. Since our goal is to find small blocks in an attempt to reduce the complexity of the last few merging operations of PBD&C, we restrict the number of diagonal blocks to be checked to 3.

When the number of matrix elements to be checked in a column of an off-diagonal block is large, the processor that receives the eigenvector information may not have enough work space to store the required rows of eigenvectors. Thus, after step 5, when the preliminary block sizes have been determined, we first check whether the work space

of each processor has enough space to accommodate eigenvectors for the sensitivity analysis. If there is not enough space, then the target block reduction is skipped and a second round of target threshold is applied with tolerance $\tau_2 \|A\|$.

4.1.3 Complexity of PBT

In the sequential BT, the computational complexity and the number of data accessed are both $O(n^2)$ [4]. In PBT, computational complexity per processor is $O(n^2/p)$; thus, the extra communication cost becomes the dominant part of the execution time since the time to transfer one floating-point number is typically much longer than to execute a floating-point operation.

Table 4.1 shows total computational and communication complexities for each step of PBT as well as the complexity of matrix redistribution between 2D and 1D at the beginning and end of PBT. In Table 4.1, nnz_1 and nnz_2 are the number of nonzero elements of matrices A' and A''', respectively (typically $nnz_1 < nnz_2 \ll n$). In Step 6, k denotes the number of matrix elements that are checked for elimination (typically $k << n$), and c is the number of processor columns in the 2D distribution of the eigenvector matrix.

As a pre-processing step for the PBD&C algorithm, the computational cost of PBT is typically minor compared to the computational cost of PBD&C. However, the scalability of the PBT algorithm may not be comparable to those algorithms with high computational complexity because of its relatively large communication overhead. We parallelize the block tridiagonalization algorithm with large application matrices in mind. Those matrices must be stored on distributed memory, and parallelization of the BT algorithm becomes essential for computing their eigensystems.

Table 4.1 Computational and communication complexities of PBT.

Steps	Comparison and local data movement	Communication cost	Addition and multiplication
0. Matrix redistribution 2D $\Leftrightarrow$ 1D	—	$2p^2\alpha + 2n^2\beta$	—
1. Global threshold	n^2/p	$2p\alpha + (n + nnz_1)\beta$	—
2. GPS reorder	$3n^{3/2}/2$ [45]	$(\alpha + n\beta)\log p$	—
3. Symmetric permutation	n^2/p	$n\alpha + n^2\beta$	—
4. Target threshold	$(n^2 - nnz_2)/p$	$(4\alpha + 4n\beta)\log p$	$(n^2 - nnz_2)/p$
5. Covering	n	—	—
6. Target block reduction	$2n$	$k(2c\alpha + 2n\beta)$	$2kn$

4.2 Parallel orthogonal block tridiagonal reduction (POBR) of dense matrix

The parallel orthogonal block tridiagonal reduction step reduces a dense matrix A to block tridiagonal form using a sequence of QR factorizations on column blocks of A, as shown in Figures 4.8 and 4.9. The resultant block tridiagonal matrix is similar to a banded matrix except that the last off-diagonal block is not a triangular.

There are parallel implementations [9, 99] of the orthogonal bandwidth reduction algorithm. The first attempt [9] of parallelization sets the restriction that the algorithmic panel width p_b, bandwidth of the reduced banded matrix bw and the block size of the 2D block cyclic matrix distribution n_b are all the same, and the matrix size n is a multiple of p_b. A later implementation [99] using PLAPACK has the flexibility of using any values for p_b, b and n_b, but the performance is not as satisfactory as the first

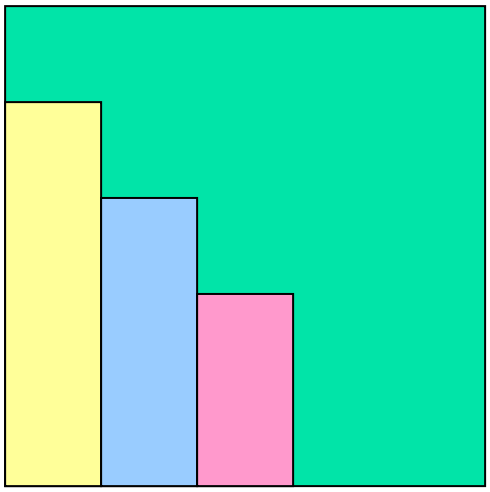

Figure 4.8 QR factorizations of column-blocks of a matrix.

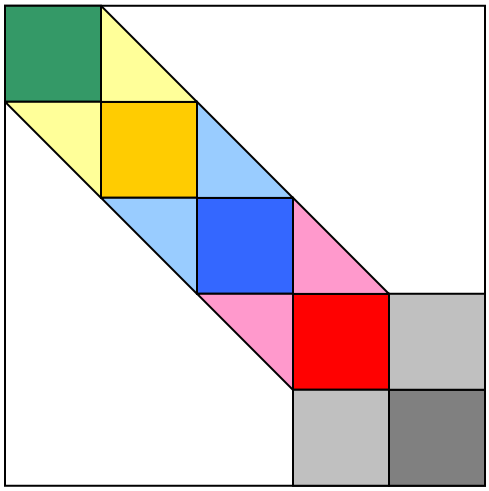

Figure 4.9 Block tridiagonal matrix after orthogonal reduction.

implementation [99]. In Section 4.2.1 we discuss how the algorithmic panel width p_b, the size of tridiagonal blocks b and the size of parallel matrix distribution block n_b are chosen for our algorithm.

4.2.1 Selection of block size *b* and panel width p_b

A critical issue in the orthogonal reduction from a dense matrix to a block tridiagonal one is how to choose the algorithmic panel width p_b of each QR factorization.

First, we consider the relationship between b and p_b. Block size b directly affects the computational complexity of the PBD&C merging operation. When b is large, the rank of the off-diagonal blocks tends to be large as well, which increases the time complexity of the PBD&C merging operation. Therefore we wish to obtain a block tridiagonal matrix with small b. However, as explained in Section 2.3.3, b should not be smaller than p_b, so we set $b = p_b$. If we choose a small panel width p_b, the resultant block size b is also small, but we may not be able to obtain full performance of level 3 BLAS operations. If p_b is large, we may obtain slightly better performance during the reduction as shown in Figure 2.20; but then b will be large and the rank of the off-diagonal blocks will likely be large as well. The reduction of execution time in POBR is not likely to compensate the increased execution time from PBD&C.

Second, we consider the relationship between n_b and p_b. Since matrices are distributed using ScaLAPACK 2D block cyclic pattern, to reduce data transfer between processor columns and the complexity of local index calculation, p_b should equal n_b, as the ScaLAPACK reduction subroutine PDSYTRD does. This guarantees that QR factorization of each matrix column block is performed on only one processor column, and does not involve row-wise communication in the processor grid.

From the above two restrictions $b = p_b$ and $p_b = n_b$, we fix the sizes of panel width and diagonal blocks of the reduced block tridiagonal matrix to be the block size of the parallel 2D block cyclic matrix distribution, i.e., $p_b = b = n_b$.

4.2.2 Complexity of parallel orthogonal reduction

To be consistent with notation used in Section 2.3, matrix column block $G_i = A_{(in_b+1:n,(i-1)n_b+1:in_b)}$ is the *i*-th panel to be factorized, and $A_i = A_{((i-1)n_b+1:n,(i-1)n_b+1:n)}$ is the lower right principal submatrix of A at the *i*-th stage of orthogonal reduction as shown in Figures 2.16 and 2.19. For convenience of reference, we replicate Figure 2.19 here as Figure 4.10. We partition A_i into 2×2 submatrix blocks:

$$A_i = \begin{array}{c} \begin{matrix} n_b & n-in_b \end{matrix} \\ \begin{bmatrix} A_{11}^i & A_{12}^i \\ A_{21}^i & A_{22}^i \end{bmatrix} \begin{matrix} n_b \\ n-in_b \end{matrix} \end{array}, \tag{4.1}$$

where $A_{21}^i = G_i$ is the submatrix to be factorized into $G_i = Q_i R_i$, and A_{22}^i is the submatrix to be updated from both sides by Q_i.

In POBR, there are four steps to compute a sequence of n_b Householder transformations and reduce column block $G_i = A_{(in_b+1:n,(i-1)n_b+1:in_b)}$ where $1 \le i \le q$ and $q = \left\lfloor \frac{n}{n_b} \right\rfloor - 1$. These four steps are:

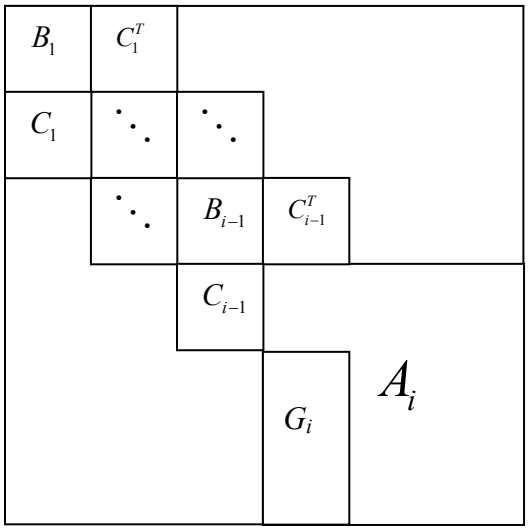

Figure 4.10 Matrix A at the *i*-th stage of orthogonal reduction.

1) Compute QR factorization of each column block G_i. The computed Householder vectors overwrite corresponding columns of G_i.

2) Construct blocked Householder transformation in the form of $I - Y_i W_i^T = H_{n_b} \cdots H_1$, where $Y_i, W_i \in R^{(n-in_b)\times n_b}$, Y_i holds columns of Householder vectors $y_j^i, 1 \le j \le n_b$, and W_i holds vectors $2y_j^i \Big/ \left(y_j^i\right)^T y_j^i$.

3) Compute submatrix $Z_i \in R^{(n-in_b)\times n_b}$ using $Z_i = A_{22}^i W_i - \frac{1}{2} Y_i W_i^T A_{22}^i W_i$ where $A_{22}^i = A_{(n_b i+1:n, n_b i+1:n)}$.

4) Apply symmetric rank-2*k* update $\tilde{A}_{22}^i = A_{22}^i - Y_i Z_i^T - Z_i Y_i^T$. A symmetric rank-2*k* update requires only half of the computation as that of a non-symmetric update, but the communication cost cannot be reduced.

The computational and communication complexity of each step is listed in Table 4.2. For a total of $k = \left\lfloor \frac{n}{n_b} \right\rfloor - 1$ blocks and $m = n - in_b$ for $1 \le i \le k$, the total floating-point operation count for all processors is

$$
\begin{aligned}
flops_{reduction} &= \sum_{i=1}^{k} 4m^2 n_b + 8mn_b^2 + 4mn_b + 2n_b^2 - 2n_b^3 \\
&= \sum_{i=1}^{k} 4(n-in_b)^2 n_b + \left(n-in_b\right)\left(8n_b^2 + 4n_b\right) + 2n_b^2 - 2n_b^3 \\
&= \frac{4}{3}n^3 - 2n^2 n_b + \frac{2}{3}nn_b^2 + 4n^2 n_b - 4nn_b^2 + 2n^2 - 2nn_b + 2nn_b - 2n_b^2 - 2nn_b^2 + 2n_b^3 \\
&= \frac{4}{3}n^3 + 2\left(n_b + 1\right)n^2 - \frac{16}{3}nn_b^2 - 2n_b^2 + 2n_b^3 \\
&= \frac{4}{3}n^3 + O\left(n_b n^2\right). \qquad (4.2)
\end{aligned}
$$

With a total of p processors, the floating-point operations executed by each processor is approximately

Table 4.2 Computational and communication complexities of POBR for reduction of one matrix column block $G_i \in \mathbb{R}^{m\times n_b}$ where $m = n - n_b i$.

Step	Computational complexity	Communication complexity
1) Compute QR factorization of G_i	$\sum_{j=1}^{n_b} 4(m-j+1)(n_b-j+1)$	$\sum_{j=1}^{n_b} (\log r)(\alpha+\beta)(n_b-j+2) + r\left[\alpha+\beta(n_b-j)\right]$
2) Construct blocked QR factorization $I - YW^T$	$\sum_{j=1}^{n_b} 4(j-1)(m-j+1)$	$\sum_{j=1}^{n_b} (\log r)(\alpha+\beta)(j-1) + r[\alpha+(j-1)\beta]$
3) Compute $Z_i = A_{22}W_i - \frac{1}{2}Y_i W_i^T A_{22} W_i$	$2m^2 n_b + 4mn_b^2 + 2mn_b$	$2(\log r)\left(\alpha + \frac{mn_b}{c}\beta\right) + (\log r) n_b^2 (\alpha+\beta) + (\log r)(\alpha + \beta n_b^2)$
4) Compute $\tilde{A}_{22} = A_{22} - Y_i Z_i^T - Z_i Y_i^T$	$2m(m+1)n_b$	$2\log r\left(\alpha + \frac{mn_b}{c}\beta\right) + 2\log c\left(\alpha + \frac{mn_b}{r}\beta\right).$
Total	$4m^2 n_b + 8mn_b^2 + 4mn_b + 2n_b^2 - 2n_b^3$	$n_b(n_b+1)(\log r)(\alpha+\beta) + 2\alpha r + n_b(n_b-1)\beta r +$ $2(\log r)\left(\alpha + \frac{mn_b}{c}\beta\right) + (\log r) n_b^2 (\alpha+\beta) +$ $(\log r)(\alpha+\beta n_b^2) + 2(\log r)\left(\alpha + \frac{mn_b}{c}\beta\right) + 2(\log c)\left(\alpha + \frac{mn_b}{r}\beta\right).$

$$flops_{reduction}^{per\ proc} \approx \frac{4n^3}{3p}\gamma . \tag{4.3}$$

The total communication cost for each processor is

$$t_{reduction}^{comm} = \sum_{i=1}^{k} n_b\left(n_b+1\right)\left(\log r\right)\left(\alpha+\beta\right)+2\alpha r+n_b\left(n_b-1\right)\beta r+4\left(\log r\right)\left(\alpha+\frac{mn_b}{c}\beta\right)+$$

$$\left(\log r\right)n_b^2\left(\alpha+\beta\right)+\left(\log r\right)\left(\alpha+\beta n_b^2\right)+2\left(\log c\right)\left(\alpha+\frac{mn_b}{r}\beta\right)$$

$$\approx \alpha\left(\frac{n}{n_b}-1\right)\left[2r+O\left(\log r\right)+O\left(\log c\right)\right]+$$

$$\beta\left[n^2\left(\frac{2\log r}{c}+\frac{\log c}{r}\right)+O\left(n\log r\right)+O\left(nr\right)\right] \tag{4.4}$$

In POBR, the number of floating-point operations in steps 3) and 4) adds up to $\frac{4}{3}n^3-\left(n_b-\frac{1}{2}\right)n^2+O\left(n\right)$. The ratio of BLAS 3 operations is then approximately $1-\frac{3n_b}{\frac{4}{3}n+2n_b}$. With fixed block size n_b for parallel data distribution, we can have more than 90% level 3 BLAS operations if $n>21n_b$. Figure 4.11 shows the ratio of level 3 BLAS operations in POBR with different matrix sizes and block sizes. Eigenvalue problems generated from application problems in scientific computing are usually very large so that $n \gg n_b$. Therefore, we are guaranteed to have high ratio of BLAS 3 operations in POBR and would expect POBR to have good performance for such matrices.

Assume the processor grid is a square grid with $r=c=\sqrt{p}$, then the estimated approximate speedup of block tridiagonal reduction can be expressed by

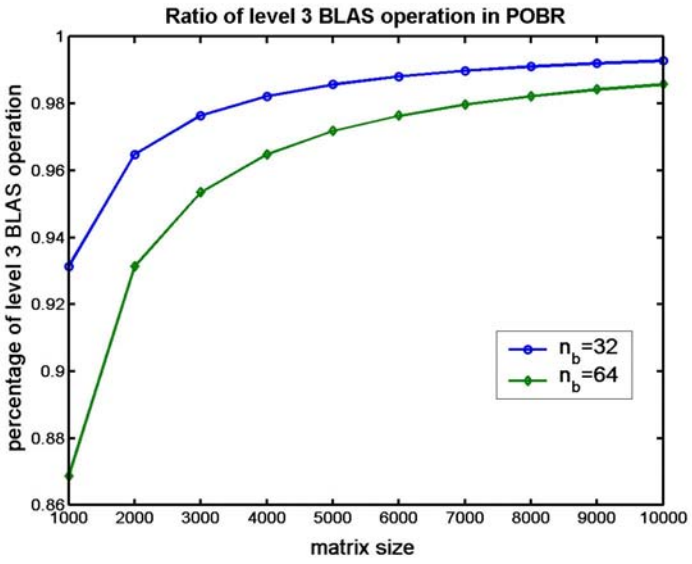

Figure 4.11 Ratio of level 3 BLAS operation in POBR, block size of parallel matrix distribution $n_b = 32, 64$.

$$Speedup(n,p) = \frac{\frac{4}{3}n^3\gamma}{\frac{4n^3}{3p}\gamma + \frac{2n\sqrt{p}}{n_b}\alpha + \frac{3n^2\left(\log\sqrt{p}\right)}{\sqrt{p}}\beta}$$

$$= \frac{p}{1 + \frac{3p\sqrt{p}\alpha}{2n^2 n_b \gamma} + \frac{9\left(\log\sqrt{p}\right)\sqrt{p}\beta}{4n\gamma}} \qquad (4.5)$$

With Equation 4.5, a theoretical speedup of POBR can be calculated if machine parameters α, β and γ are known. On Cheetah, we have $\alpha = 7\mu s$ and $\beta = 5.7ns$ from message passing latency and bandwidth benchmarking results [38] and $\gamma = 0.315ns$ from performance test of vendor optimized matrix multiplication subroutine DGEMM [38]. The theoretical speedup curves are shown in Figure 4.12 with $n_b = 32$.

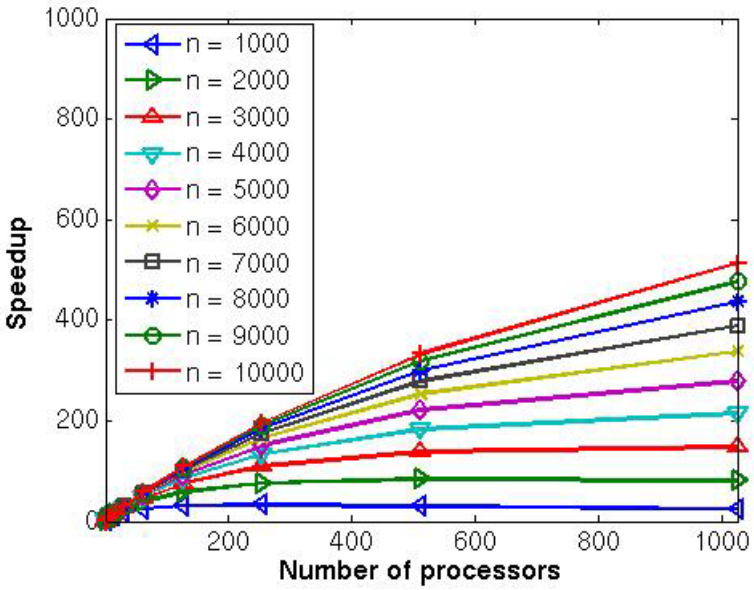

Figure 4.12 Theoretical speedup model of POBR.

5 Numerical results

In this section, we present results of accuracy and performance tests. Our tests were run on the IBM p690 system nicknamed Cheetah in Oak Ridge National Laboratory. System specifications and important benchmarking results are listed in Table 5.1. The performance of the parallel block tridiagonal divide-and-conquer subroutine PDSBTDC is compared to the ScaLAPACK divide-and-conquer subroutine PDSYEVD [92]; the performance of parallel orthogonal block tridiagonal reduction subroutine PDSBTRD is compared to the ScaLAPACK symmetric tridiagonalization subroutine PDSYTRD [21]. Parallel block tridiagonalization subroutine PDSBTRI is tested separately. Finally the performance of the parallel approximate eigensolver which uses PDSBTDC, PDSBTRI and PDSBTRD as core components is tested using application matrices with different structures.

The Fortran compiler on Cheetah is IBM's `xlf` version 8.1. Codes were compiled in the default 32-bit compile mode and linked to the 32-bit PESSL library [56] which includes the vendor optimized version of BLAS . The compiler options used are:

```
-O4 -qarch=auto -qcache=auto –qtune=auto
-bmaxdata:0x70000000.
```

For the computed eigensolutions of a real symmetric matrix $A = \hat{X}\hat{\Lambda}\hat{X}^T$ where $\hat{X}$ is the computed approximate eigenvector matrix and $\hat{\Lambda}$ is the diagonal matrix that contains the computed approximate eigenvalues, we use the scaled residual error

$$\mathcal{R} = \max_{i=1,\cdots,n} \frac{\left\| A\hat{x}_i - \hat{\lambda}_i \hat{x}_i \right\|_2}{\left\| A \right\|_2}$$

and the scaled departure from orthogonality

$$\mathcal{O} = \frac{\max_{i=1,\cdots,n} \left\| \left(\hat{X}^T \hat{X} - I \right) e_i \right\|_2}{n}$$

to evaluate the accuracy of results.

For all the numerical tests, the number of processors used is a power of 2. We start from the smallest number of processors that provides sufficient memory to solve the

Table 5.1 Cheetah system specifications and benchmarks [38].

Number of nodes		27
Memory per node		32 GB for most of the nodes
Processors per node		32
CPU frequency		1.3 GHz
L1 cache	Data	32 KB
	Instruction	64 KB
L2 cache		1.5 MB shared between 2 processors
L3 cache		32 MB off chip
Interconnect switch		Federation
Message passing latency		7 μs
Message passing bandwidth		1400 MBs
DGEMM GFLOPS per processor		3.174 GFLOPS

problems in parallel and verify computational results, and increment the number of processors up to 512.

5.1 Test matrices

There are three types of matrices in our tests: 1) LAPACK/ScaLAPACK test matrices with different eigenvalue distributions [27], 2) matrices generated from application problems in quantum chemistry and condensed matter physics, and 3) random matrices. Some of those matrices are banded or block tridiagonal, some are dense but "effectively" sparse, and some are dense without any specific structure. Matrix sizes range from 3,014 to 20,000. In this section, we present representative performance and accuracy results. The complete set of numerical test results is given in the **Appendix**.

5.1.1 LAPACK/ScaLAPACK test matrices

A banded matrix is a special form of block tridiagonal matrix in that all off-diagonal blocks are triangular. We use banded matrices with different eigenvalues distributions generated by LAPACK subroutine DLATMS to test PDSBTDC. Since the computational complexity of PDSBTDC increases on the order of n^3 with the rank of the off-diagonal block for the final merging operation (see Section 2.1.4), we limit the bandwidth of test matrices to 20, so that the ranks of the off-diagonal blocks are never greater than 20. There are six types of matrices in this category with different eigenvalue distributions. For each type, test matrices are generated for five different sizes: 4,000, 8,000, 12,000, 16,000 and 20,000.

P-clu0. Eigenvalues clustered at $\pm\varepsilon_{mach}$, only one eigenvalue is ± 1.

P-clu1. Eigenvalues clustered at ± 1, only one eigenvalue is $\pm\varepsilon_{mach}$.

P-geom. Eigenvalues distributed in a geometric sequence ranging from 1 to ε_{mach} with random signs attached to eigenvalues, $\lambda_i = \pm\left(\varepsilon_{mach}\right)^{i-1/n-1}$.

P-arith. Eigenvalues distributed in an arithmetic sequence ranging from 1 to ε_{mach} with random signs attached to eigenvalues, $\lambda_i = \pm\left[1-\left(1-\varepsilon_{mach}\right)\left(i-1\right)/\left(n-1\right)\right]$.

P-log. Logarithm of eigenvalues uniformly distributed in the range from 1 to ε_{mach} with random signs attached to eigenvalues.

P-rand. Random eigenvalues uniformly distributed in $\left(-1,1\right)$.

5.1.2 Application matrices

In this section we give a brief description of test matrices generated from the calculation of the electronic structure for different types of molecules. For each type of molecule, different test matrices are generated, typically by incorporating different number of molecules in the model. However, their Fock matrices and eigenvalue distributions look very similar, except that matrix sizes are different. In most of our tests, we only test the largest matrix from a molecule family unless otherwise specified.

A-alk. Alkane. Matrices are generated from simulating alkane molecules using the CNDO method [81, 82, 83]. The general molecular formula of an alkane is C_nH_{2n+2}. Figures 5.1 and 5.2 show the magnitudes of elements of a Fock matrix generated from $C_{502}H_{1006}$ and its eigenvalue distribution. The size of the matrix is 3,014.

A-ala. Polyalanine. Matrices are generated from simulating polypeptide molecules made from alanine using the MNDO method [28]. Figures 5.3 and 5.4 show the magnitudes of elements of the Fock matrix from a linear polyalanine chain of length 200 used in our test and its eigenvalue distribution. All matrices in this category are banded matrices. The matrix used in our tests is of size 5,027, and its bandwidth is 79.

A-Si. Silicon crystal. Matrices are generated from simulating silicon crystals using the PBE [79] functional in density functional theory with differing number of unit cells containing 8 atoms each. Figure 5.5 shows the magnitudes of elements in the matrix used in our tests with 5 unit cells in the x direction, and 4 in both the y and z directions. Figure 5.6 gives its eigenvalue distribution. The size of this matrix is 8,320.

A-tPA. Trans-Polyacetylene (PA). Trans-PA consists of a chain of CH units. It has the general molecular formula trans-$(CH)_n$. The SSH Hamiltonian [90], which is a tight-binding approximation and includes only the nearest neighboring atoms, is combined with the Hartree-Fock approximation to produce test matrices in this family. Figures 5.7 and 5.8 show the magnitudes of matrix elements of trans-$(CH)_{8000}$ and its eigenvalue distribution. Matrices used in our tests are generated from trans-$(CH)_{8000}$ and trans-$(CH)_{16000}$, and the sizes of the corresponding matrices are 8,000 and 16,000, respectively.

5.1.3 Random matrices

There are two types of random matrices in our tests. Each random matrix element is generated by the C built-in random number generator. For each type, five different matrices are again generated with sizes: 4,000, 8,000, 12,000, 16,000 and 20,000.

R-bt. Random symmetric block tridiagonal matrices of block size 20. These matrices are used to test the parallel block tridiagonal divide-and-conquer subroutine PDSBTDC.

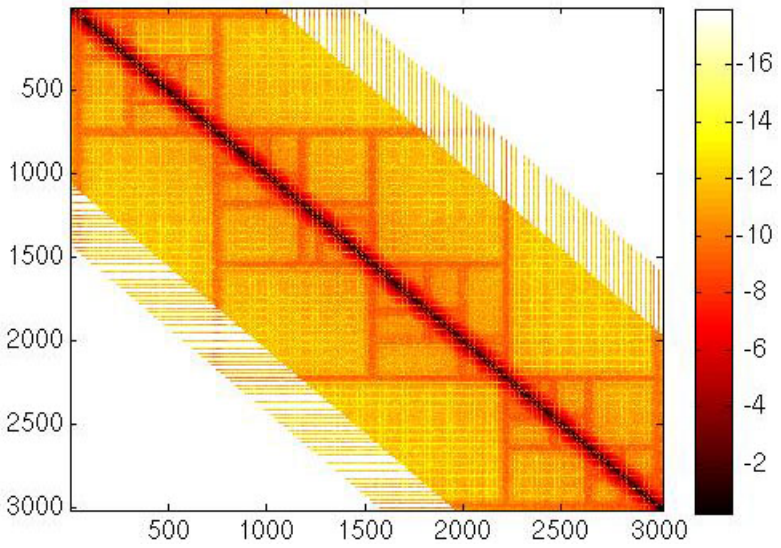

Figure 5.1 $\log_{10}$ of absolute value of matrix elements for alkane $C_{502}H_{1006}$ molecule, $n = 3,014$.

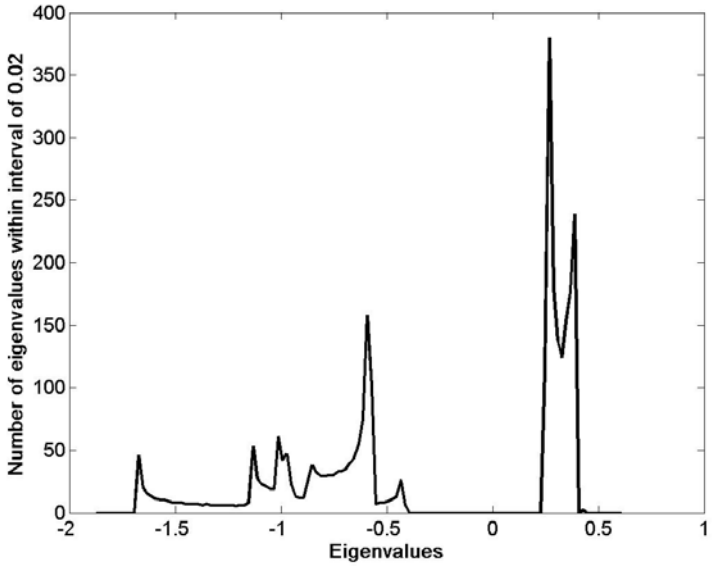

Figure 5.2 Eigenvalue distribution of matrix in Fig. 5.1.

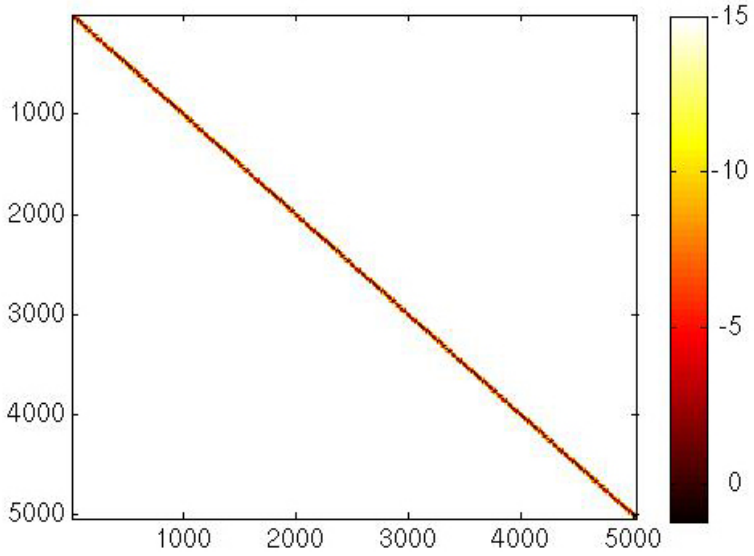

Figure 5.3 $\log_{10}$ of absolute value of matrix elements for linear polyalanine chain of length 200, $n = 5,027$.

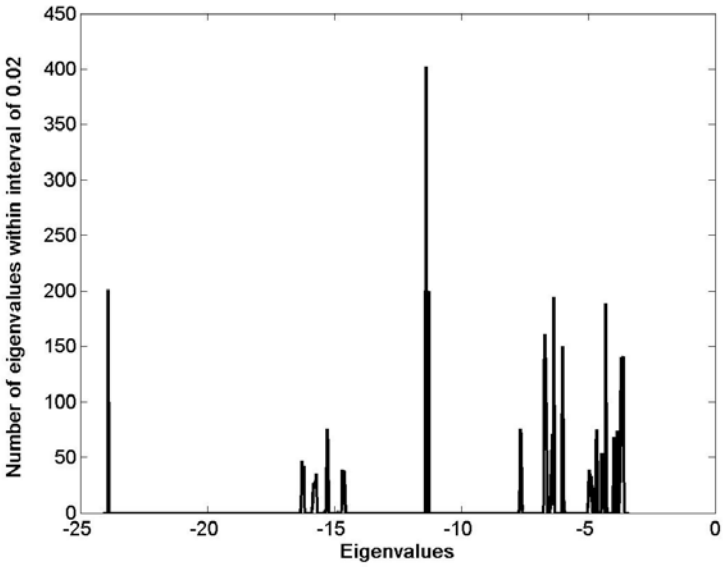

Figure 5.4 Eigenvalue distribution of matrix in Fig. 5.3.

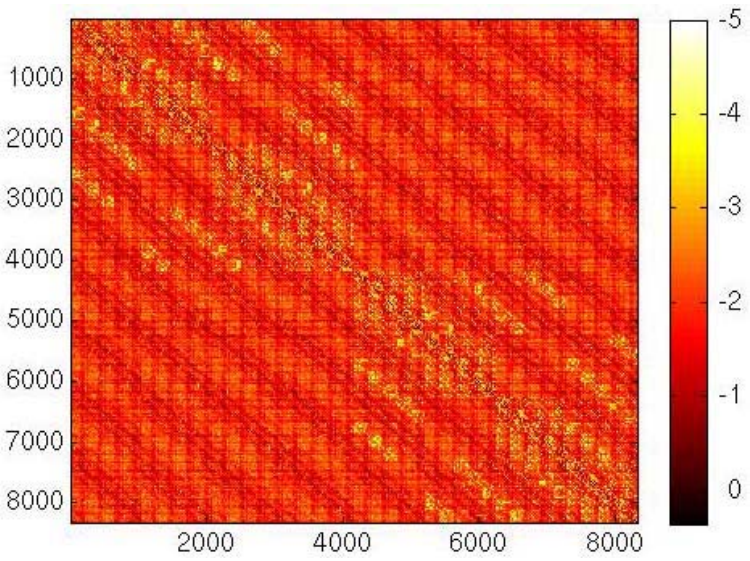

Figure 5.5 $\log_{10}$ of absolute value of matrix elements for silicon crystal molecule, $n = 8,320$.

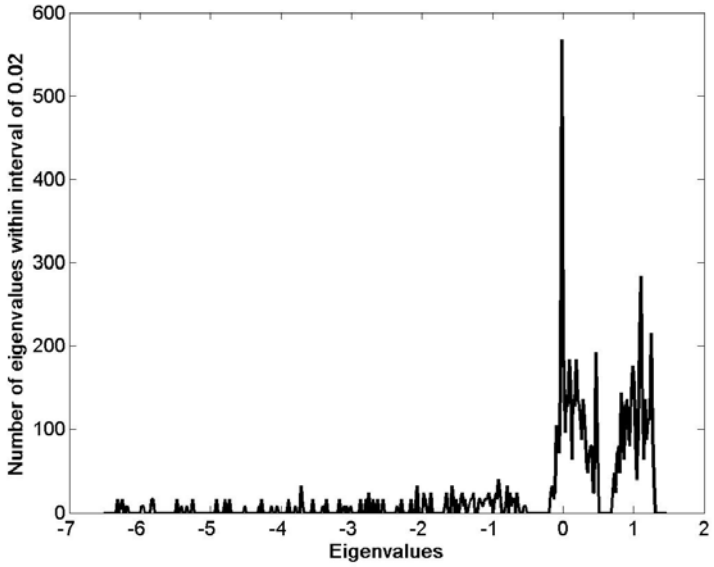

Figure 5.6 Eigenvalue distribution of matrix in Fig. 5.5.

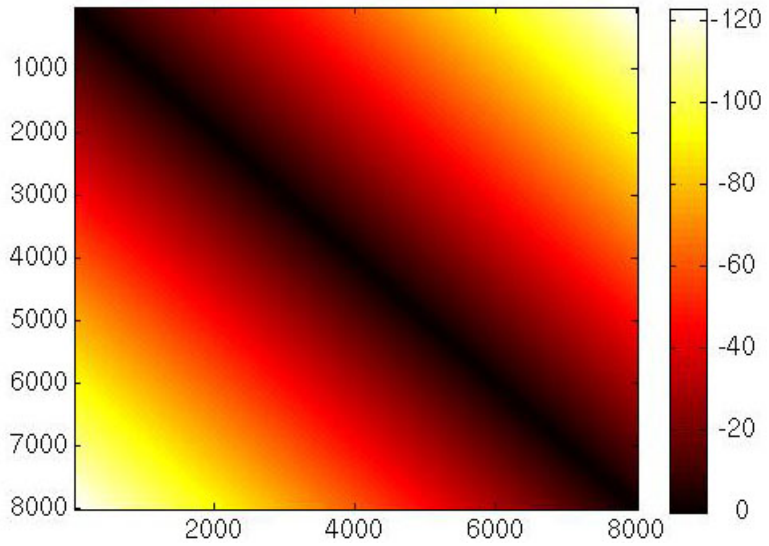

Figure 5.7 $\log_{10}$ of absolute value of matrix elements for trans-PA molecule, $n = 8,000$.

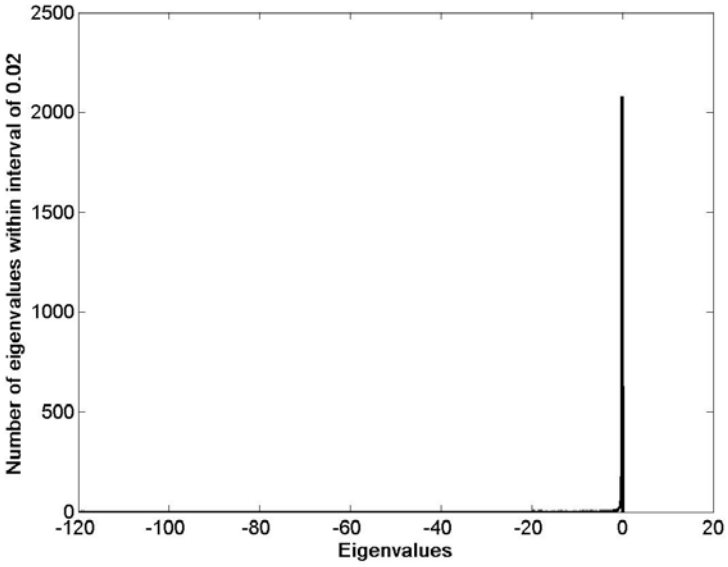

Figure 5.8 Eigenvalue distribution of matrix in Fig. 5.7.

R-den. Random symmetric full matrices for the test of the parallel orthogonal block tridiagonal reduction subroutine PDSBTRD.

5.2 Test results for PBD&C subroutine PDSBTDC

In the tests of PDSBTDC, we use block tridiagonal matrices **P-clu0, P-clu1, P-geom, P-arith, P-log, P-rand** and **R-bt**. Five matrices of each type are tested with different orders: 4,000, 8,000, 12,000, 16,000 and 20,000; the block size on each is 20. We also use application matrix **A-ala**, which has a matrix size of 5,027 and the block sizes of 104 for the first and the last blocks and 79 for other diagonal blocks. The execution times of PDSBTDC are scaled by the execution times of the ScaLAPACK divide-and-conquer subroutine PDSYEVD.

First we set the accuracy tolerance to 10^{-6} for PDSBTDC. Figure 5.9 shows the relative execution time of PDSBTDC to ScaLAPACK subroutine PDSYEVD in log scale using **P-geom** matrices — eigenvalues with geometric distribution. Figure 5.10 shows the maximum residual $\mathcal{R}$ and orthogonality errors $\mathcal{O}$ over all five **P-geom** matrices. Under the stated accuracy tolerance, all the block tridiagonal matrices in **P-geom** have rank of 0 for the off-diagonal block of the final merging operation, which decouples the problem into two smaller ones. In addition, matrices with clustered eigenvalues tend to have very high ratio of deflation (see Figure 2.4). Those two factors lead to the high efficiency of PDSBTDC. Matrices **P-clu0**, **P-clu1** and **P-log** all have clustered eigenvalues and display performance similar to that shown in Figure 5.9.

Figure 5.11 shows the performance of PDSBTDC on **P-arith** matrixes. As the eigenvalues are evenly distributed, deflation rate decreases. Another factor that contributes to the slower performance of PDSBTDC is that the ranks of the off-diagonal blocks in the **P-arith** matrices are much higher than those in other matrix types mentioned in preceding paragraph. Performance of the random block tridiagonal matrices **R-bt** is similar to that of **P-arith**. Performance of matrices with random eigenvalues **P-rand** is slower than that of **P-geom** (Figure 5.9), but much better than that of **P-arith** (Figure 5.11). Maximum residual $\mathcal{R}$ and orthogonality errors $\mathcal{O}$ for matrices **P-arith** are

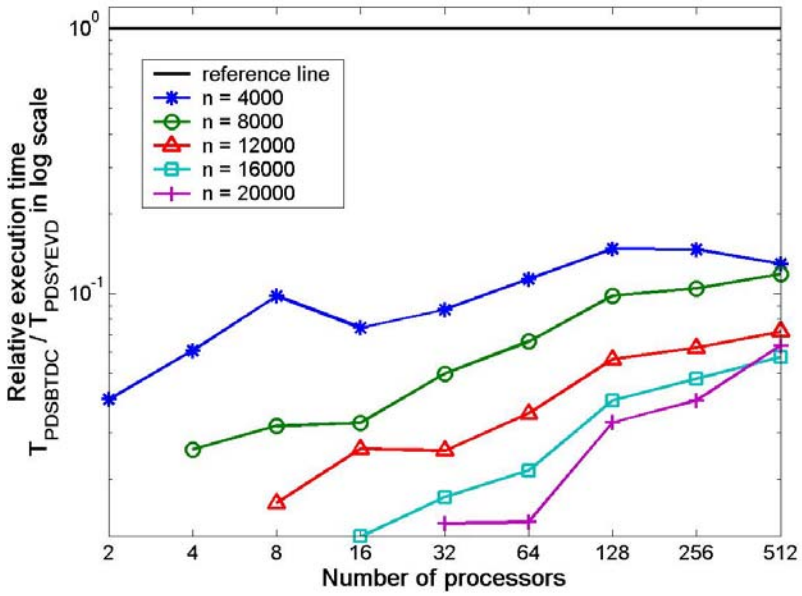

Figure 5.9 Execution time of PDSBTDC relative to PDSYEVD in log scale using **P-geom** matrices.

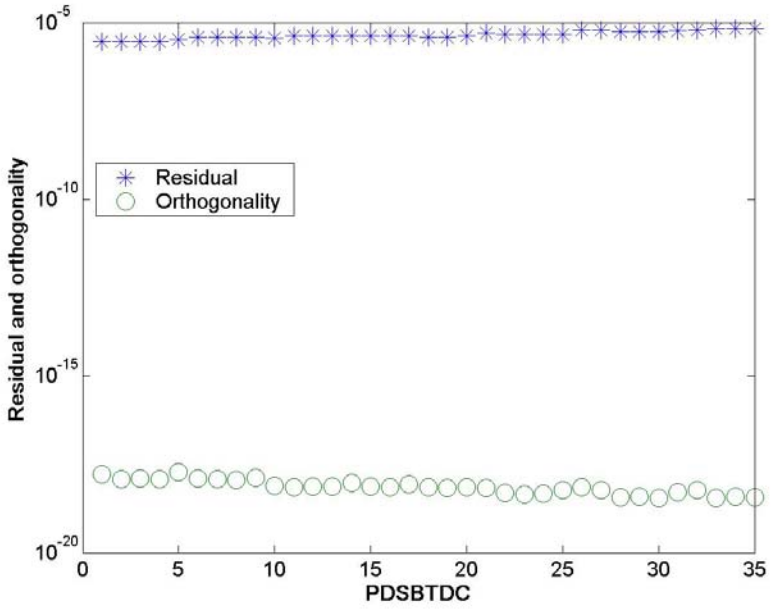

Figure 5.10 Maximum residual and orthogonality error for PDSBTDC on **P-geom** matrices.

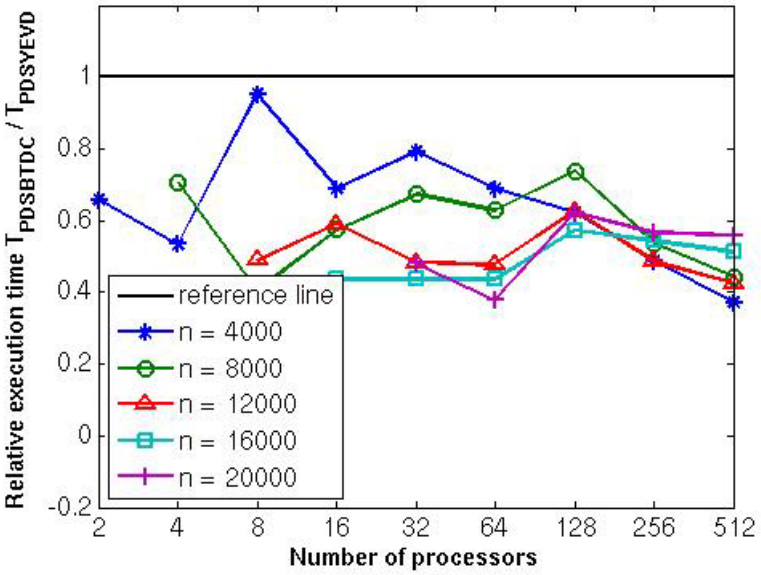

Figure 5.11 Execution time of PDSBTDC relative to PDSYEVD using **P-arith** matrices.

displayed in Figure 5.12.

To test performance and accuracy of PDSBTDC with different accuracy requirements, we use the application matrix **A-ala** and set the accuracy tolerance to different values: 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10} and 10^{-12}. Figure 5.13 shows that as the tolerance decreases, execution time increases due to less deflation and higher ranks for off-diagonal blocks. For example, with a tolerance of 10^{-6}, the ranks of the off-diagonal blocks range from 20 to 21, and a very high deflation rate (about 90%) in the last 3 merging operations significantly reduces the total amount of computation.

Since most of the matrices in our test are too large to be computed using one processor, it is not feasible to measure the speedup of PDSBTDC using the traditional definition, i.e., $speedup = \frac{T_s}{T_p}$ where T_s is the time to run the fastest sequential code and T_p is the time to run the parallel code with p processors. In Figure 5.14, we use a smaller matrix to

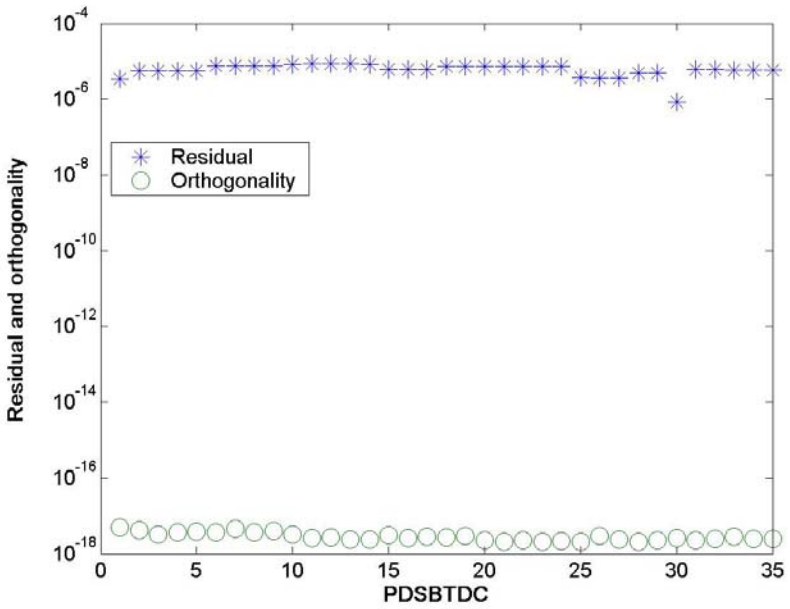

Figure 5.12 Maximum residual and orthogonality error of PDSBTDC on **P-arith** matrices.

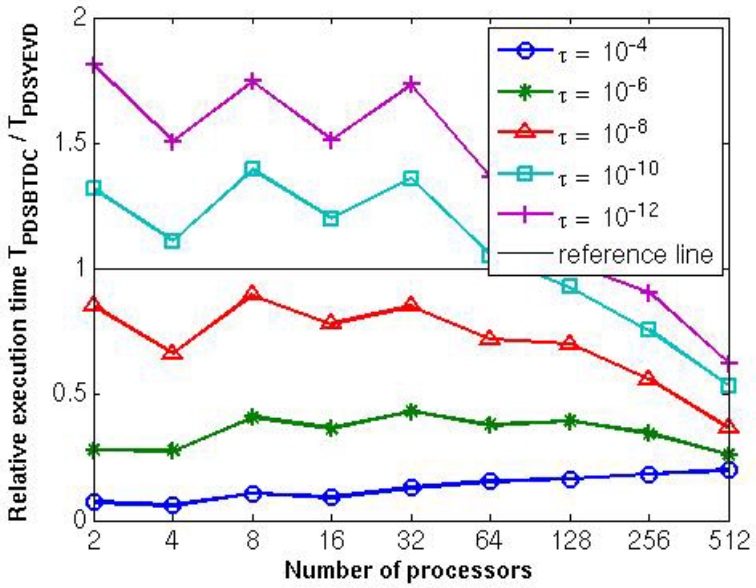

Figure 5.13 Execution time of PDSBTDC relative to PDSYEVD using application matrix **A-ala** with different accuracy tolerance.

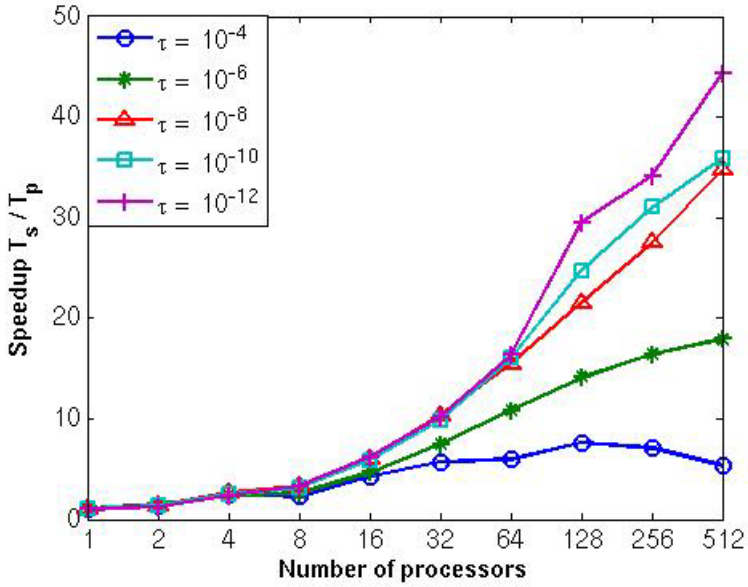

Figure 5.14 Speedup of PDSBTDC using matrix **A-ala** with tolerances $\tau = 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}$, matrix size $n = 5,027$.

evaluate speedup with different accuracy requirements since computational complexity increases as accuracy tolerance becomes smaller. Speedup factors should be significantly better on larger, more appropriately sized matrices.

5.3 Test results for POBR subroutine PDSBTRD

Random matrices are used to test performance of the parallel orthogonal block tridiagonal reduction subroutine PDSBTRD. Figure 5.15 shows the execution times of PDSBTRD scaled by that of the ScaLAPACK subroutine PDSYTRD. Performances of both subroutines scale up with the number of processors in use. It should be noted that the floating-point operation count for the two subroutines are not the same since PDSBTRD only reduces a matrix to block tridiagonal form while PDSYTRD reduces a matrix to tridiagonal form. The improved performance as a result of using level 3 BLAS operations can be seen from the relative execution time of PDSBTRD to PDSYTRD. In particular,

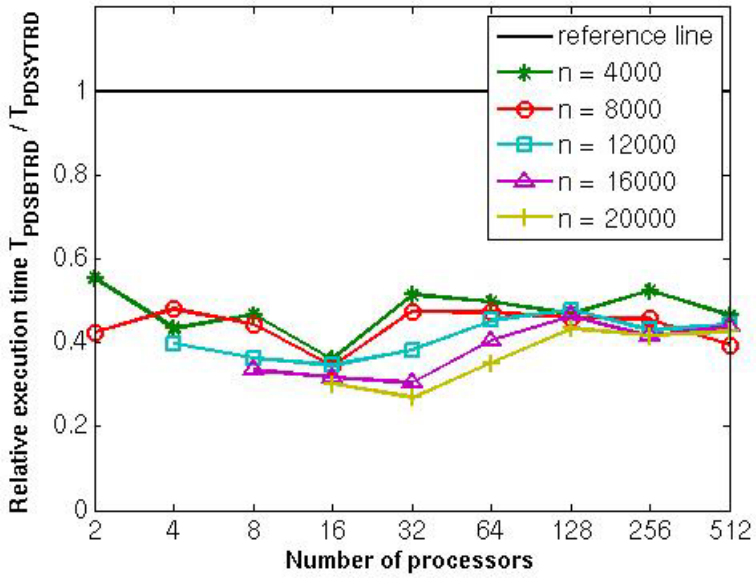

Figure 5.15 Relative execution time of PDSBTRD to PDSYTRD using random matrices **R-den**.

PDSBTRD performs better when the problem size per processor n^2/p becomes larger, which matches **Corollary 2.2**.

5.4 Test results for PBT subroutine PDSBTRI

In our application matrices, the alkane and trans-PA matrices have strong locality property, that is, the larger elements are close to the diagonal and the magnitudes of matrix elements decrease as they move away from the diagonal. In an iterative method like the SCF, a non-linear eigenvalue problem is solved by solving a linear eigensystem iteratively until convergence. For the alkane matrix **A-alk** of size 3014, we completed all the iterations using a sequential SCF subroutine and stored the Fock matrices and eigenvector matrices from each iteration. Thus, in the test of PDSBTRI using matrix **A-alk**, we are able to test the optional target block reduction (PBT) step using the eigenvector matrix from the previous iteration as approximate eigenvectors. The trans-PA matrices **A-tPA** of sizes 8,000 and 16,000 are used for the test of PDSBTRI without PBT.

Figure 5.16 shows the execution times of PDSBTRI with tolerance $\tau = 10^{-6}$. The parallel block tridiagonalization algorithm contains steps that are sequential in nature as well as steps that parallelize well. For example, the matrix reorder (*step 2*) is completely sequential; while the global thresholding (*step 1*) and the modified target thresholding (*step 4*) are embarrassing (or pleasantly) parallel, in that each processor checks elements to be eliminated independently. As the number of processors increases, the execution times for steps 2 and 4 decrease. However, the overhead of redistributing the matrix from 2D block cyclic distribution to 1D column block distribution increases with the number of processors. Therefore, the execution time of PDSBTRI on Cheetah remains almost constant as the number of processors increases. Overall, the time for block tridiagonalization is still very small compared to the time for solving eigenproblem.

We compare the eigenvalues of the block tridiagonalized matrix M and the original matrix A computed to full accuracy. Table 5.2 shows that the errors in eigenvalues are

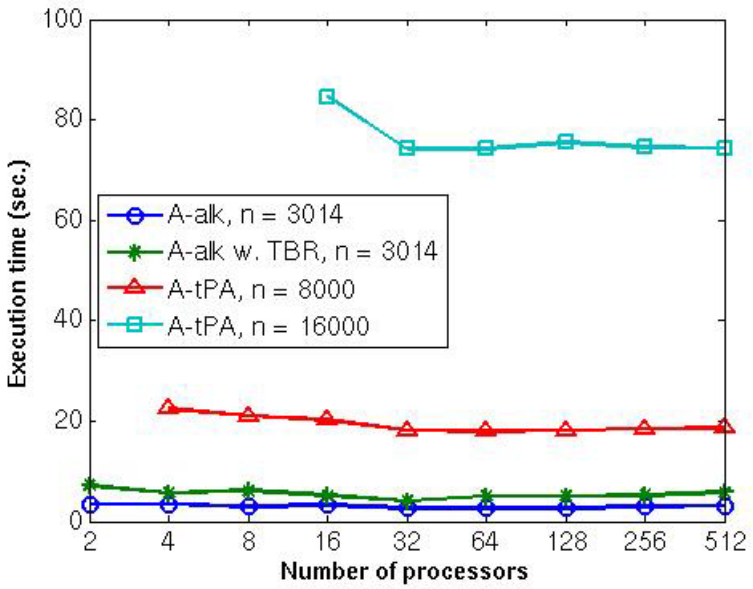

Figure 5.16 Execution time of PDSBTRI using matrices **A-alk** and **A-tPA** with $\tau = 10^{-6}$.

Table 5.2 Scaled eigenvalue error $\left|\lambda(A)-\lambda(M)\right|/\|A\|$.

Matrix	**Size**	**With TBR**	**Tolerance**	$\left\|\lambda(A)-\lambda(M)\right\|/\|A\|$
A-alk	3,014	No	10^{-6}	3.55×10^{-7}
		Yes	10^{-6}	3.57×10^{-7}
A-tPA	8,000	No	10^{-6}	8.04×10^{-8}
	16,000	No	10^{-6}	7.19×10^{-8}

bounded by $\tau\|A\|$ (see Appendix for performances and eigenvalue errors with tolerances 10^{-4} and 10^{-8}).

5.5 Test of parallel approximate eigensolver

As we described at the beginning of Section 1, our goal is to develop a parallel eigensolver that computes approximate eigenpairs of a real symmetric matrix. This eigensolver chooses eigen-decomposition algorithms based on matrix structure and the accuracy requirement. The central parts of this approximate eigensolver are subroutines PDSBTDC, PDSBTRD and PDSBTRI. In Section 5.5.1 we describe the structure of our approximate eigensolver. Test results of the approximate eigensolver using application matrices are shown in Section 5.5.2.

5.5.1 Structure of parallel approximate eigensolver

Given a real symmetric matrix $A \in \mathbb{R}^{n\times n}$ and accuracy requirement $\varepsilon_{mach} \le \tau < 0.1$, the approximate eigensolver determines what algorithm to use to compute all eigenpairs of A as the flow chart in Figure 5.17 shows:

1) If the accuracy requirement is high, then ScaLAPACK subroutine PDSYEVD is used to compute eigenpairs of A to full accuracy.

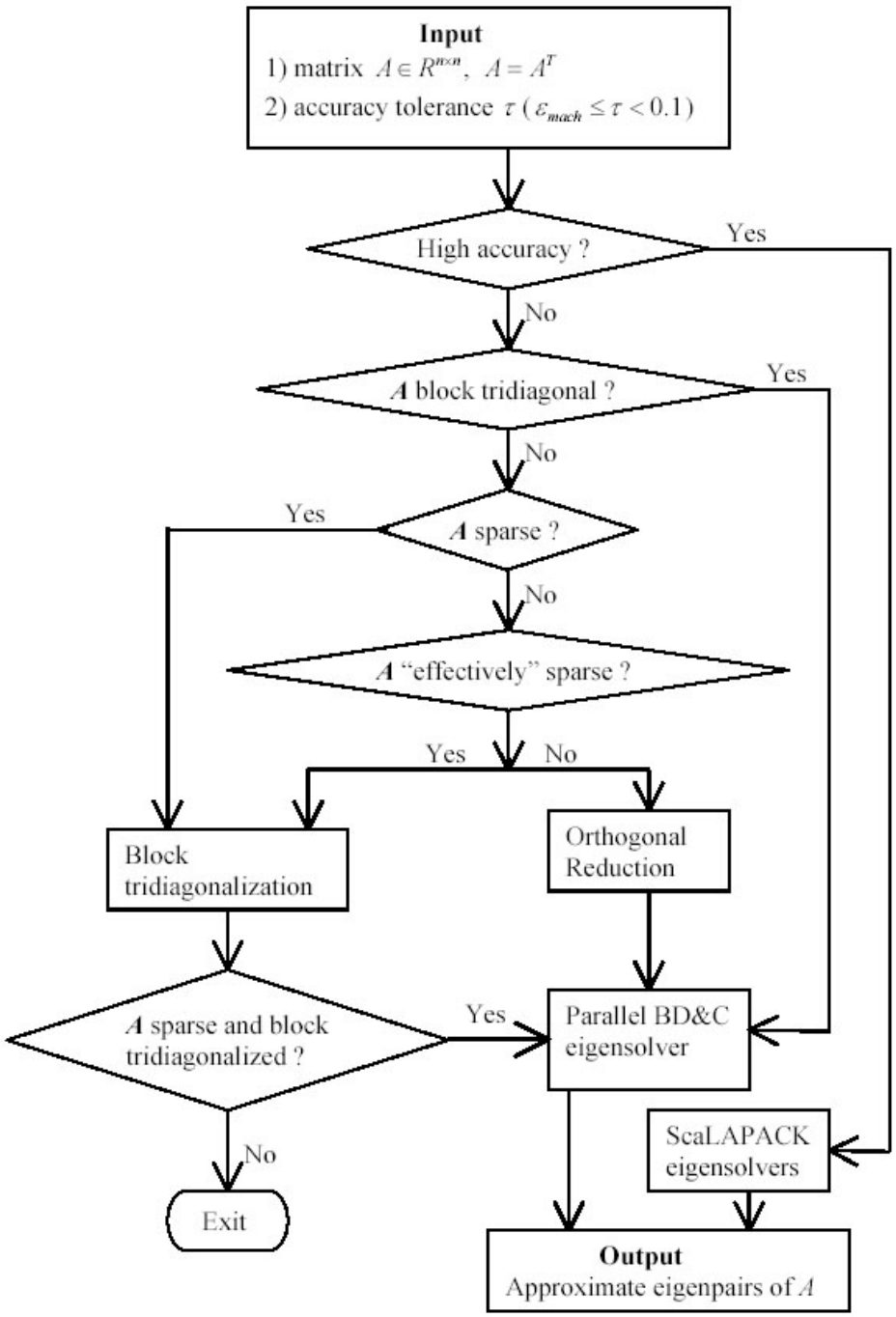

Figure 5.17 Structure of parallel approximate eigensolver.

2) If the accuracy requirement is low and A is sparse or "effectively" sparse, PDSBTRI is used to transform A into block tridiagonal matrix M. After that, PDSBTDC is used to compute approximate eigenpairs of M.
3) If the accuracy requirement is low and A does not have any structure, i.e., A is not block tridiagonal or cannot be transformed into block tridiagonal matrix using PDSBTRI, then A is reduced to block tridiagonal matrix M using orthogonal block tridiagonal reduction subroutine PDSBTRD. Then PDSBTDC is used to decompose M. Finally, the eigenvector matrix of M is back transformed to the eigenvector matrix of A.

5.5.2 Numerical tests of parallel approximate eigensolver

All the decision-making steps in the approximate eigensolver are heuristic and do not have an exact and unique solution. For example, when is an accuracy tolerance regarded as "high accuracy," and what matrix can be regarded as "effectively" sparse. Our user interface provides the option for the user to input information about the matrix structure and accuracy requirement. If the user knows the structure of the input matrix in advance, he may provide this information. Otherwise, the approximate eigensolver uses a heuristic method to determine what algorithms to use.

In our numerical tests, we used $\tau = 10^{-6}$ as a threshold for the accuracy requirement, i.e. when $\tau < 10^{-6}$, we compute eigenpairs of A to full accuracy; otherwise we compute eigenpairs of A to the required low accuracy. Application matrices **A-alk**, **A-ala**, **A-Si** and **A-tPA** are used to test the performances of PDSBTDC, PDSBTRI and PDSBTRD working together as a whole package. Matrices **A-alk** and **A-tPA** are "effectively" sparse matrices, thus PDSBTRI is used followed by PDSBTDC. Matrix **A-ala** is a block tridiagonal matrix, therefore PDSBTDC can be directly applied to it. Matrix **A-Si** does not have any usable structure, so it is first reduced to block tridiagonal form with block size $b = 32$, then solved by PDSBTDC followed by back transformation.

With a tolerance $\tau = 10^{-6}$, Figure 5.18 shows that PDSBTDC is very efficient for matrices **A-ala** and **A-tPA**, because **A-ala** has a very high ratio of deflation although the ranks of off-diagonal blocks are high, and all the off-diagonal blocks in **A-tPA** have very low ranks. The approximate eigensolver does not perform well on matrix **A-alk** due to its relatively low ratio of deflation. The off-diagonal blocks in matrix **A-Si** have full rank of 32 after orthogonal reduction, and the merging operations suffer from a low deflation rate. Those two factors lead to slow execution though the accuracy tolerance is relatively large. For matrices **A-alk** and **A-Si**, we further reduce the accuracy to 10^{-4} but leave the block size for **A-Si** at 32. Figure 5.19 shows that PDSBTRI followed by PDSBTDC performs much better on **A-alk**. The improvement of performance is a result of lower ranks for off-diagonal blocks and a higher ratio of deflation. With matrix **A-Si**, the ranks of the off-diagonal blocks remain unchanged after orthogonal reduction. Although there is a significant improvement in performance due to a higher ratio of deflation, it is still slower than PDSYEVD due to its high computational complexity introduced by the high ranks of the off-diagonal blocks. With tolerance $\tau = 10^{-4}$, we reduce the block sizes b of **A-Si** to 16, so that the ranks of all off-diagonal blocks are no greater than 16. Compared to $b = 32$, there is a small amount of performance loss in PDSBTRD (less than 5%); but the ranks of all off-diagonal blocks are reduced by half, which leads to approximately 50% reduction in execution time in PDSBTDC. Figure 5.19 shows that the approximate eigensolver is very competitive when block size and tolerance are set to $b = 16$ and $\tau = 10^{-4}$, respectively. However, with block sizes smaller than 16, the effect of level 3 BLAS operations is significantly reduced and the performance of the parallel approximate eigensolver is also degraded.

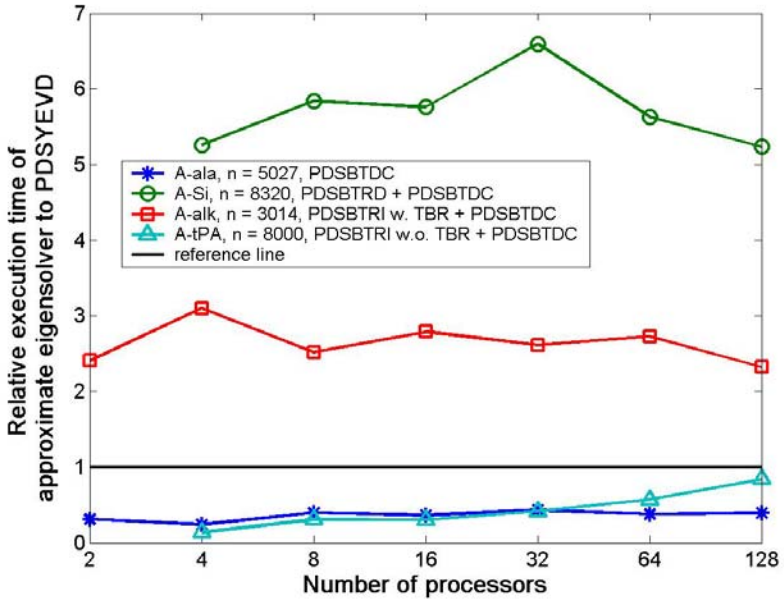

Figure 5.18 Relative execution times of approximate eigensolver to PDSYEVD using matrices **A-alk**, **A-ala**, **A-Si** and **A-tPA** with $\tau = 10^{-6}$.

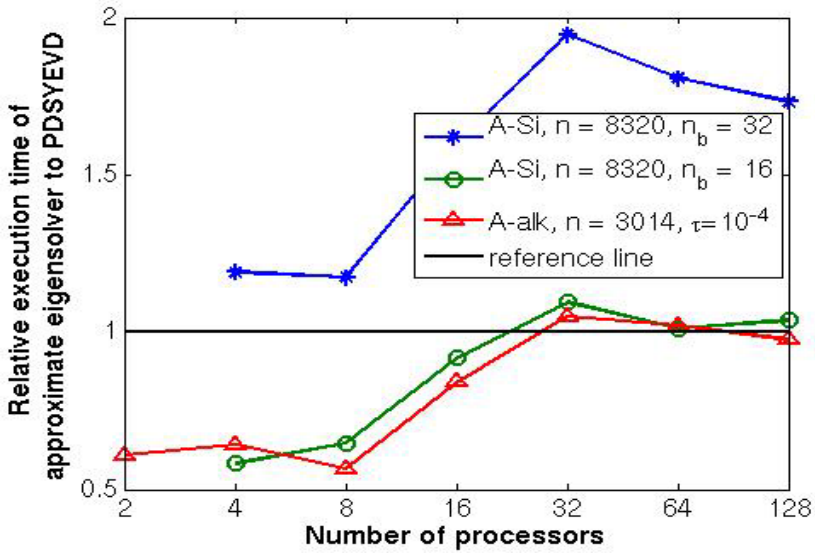

Figure 5.19 Relative execution times of approximate eigensolver to PDSYEVD using matrices **A-alk** and **A-Si** with $\tau = 10^{-4}$. For matrix **A-Si**, block sizes are 16 and 32.

6 Conclusion

In conclusion, this dissertation addresses several efficient algorithms for a parallel approximate eigensolver for real symmetric matrices. Given a real symmetric matrix A and an accuracy parameter τ, the approximate eigensolver computes the approximate eigensolutions of A such that $\left\| A - X\Lambda X^T \right\|_2 = O\left(\tau \|A\|_2\right)$ and $\max_{i=1,\cdots,n} \left\| \left(XX^T - I \right) e_i \right\|_2 = O\left(\varepsilon_{mach} n\right)$, where X is the approximate eigenvector matrix and Λ is the diagonal matrix that contains the approximate eigenvalues.

The three major algorithms in this approximate eigensolver are: 1) parallel block tridiagonal divide-and-conquer algorithm (subroutine PDSBTDC); 2) parallel orthogonal block tridiagonal reduction algorithm (subroutine PDSBTRD); and 3) parallel block tridiagonalization algorithm (subroutine PDSBTRI). Based on the matrix structure and accuracy requirement, the approximate eigensolver chooses proper combination of algorithms to compute efficiently all eigenvalues and eigenvectors of a real symmetric matrix to prescribed accuracy. If high accuracy is required, the eigensolver chooses PDSYEVD in ScaLAPACK to compute eigensolutions to full accuracy. On the other hand, if low accuracy is sufficient, depending on matrix structure, a proper combination of the above three subroutines is selected.

Complexity analyses and numerical tests show that for a low accuracy such as $\tau = 10^{-6}$, PDSBTDC is very efficient on block tridiagonal matrices with either relatively low ranks for off-diagonal blocks or very high deflation rate during the merging operations, or both.

Traditional eigensolvers for real symmetric dense matrices compute all eigenvalues and eigenvectors in three steps: 1) reduction to tridiagonal form; 2) decomposition of tridiagonal matrix; and 3) back transformation. It has been shown that the reduction step is the most time consuming step [95] because of its high ratio of level 2 BLAS operations. Although algorithms for real symmetric tridiagonal eigenvalue problems have been intensively studied and improved, the execution time for orthogonal reduction to tridiagonal form dominates the total execution time. The parallel block tridiagonal divide-

and-conquer algorithm does not require this reduction-to-tridiagonal step. Instead, it solves the eigenproblem either directly or after reduction to block tridiagonal form. A mixed data/task parallel implementation maintains workload balance and achieves good speedup. However, when the rank of the off-diagonal block for the final merging operation is large, say exceeds 20, and the deflation rate is low, then PDSBTDC is no longer competitive due to its high computational complexity.

When the input matrix is sparse or "effectively" sparse, we use the block tridiagonalization subroutine PDSBTRI to construct a block tridiagonal matrix M that is a sufficiently accurate approximation to the original input matrix A. The execution time of PDSBTRI is usually negligible compared to the execution time of the eigen-decomposition of the resultant block tridiagonal matrix. When the combination of PDSBTRI followed by PDSBTDC is used, its performance behaves similarly to that of PDSBTDC.

When the input matrix is dense and has no specific structure, the parallel orthogonal block tridiagonal reduction subroutine PDSBTRD followed by subroutine PDSBTDC is used. PDSBTRD is very efficient by itself due to its high ratio of Level 3 BLAS operations in the algorithm; however, the off-diagonal blocks tend to have full ranks even when low accuracy is required. Since the block size of the block tridiagonal matrix equals the block size of the parallel 2D matrix distribution, which is typically 32 in PDSBTRD, each off-diagonal block usually has a full rank of 32. Reducing the accuracy requirement increases the deflation rate, but typically does little to reduce the ranks. One may try to reduce the ranks of off-diagonal blocks using smaller block size for parallel matrix distribution. For our tests using a block size of 16, the effect of level 3 BLAS operations in PDSBTRD is reduced and the frequency of data communication is increased, but the performance improvement in PDSBTDC is great enough to compensate the small amount of performance loss in PDSBTRD.

In general, the parallel approximate eigensolver is efficient and accurate to the prescribed tolerance. The time required for computing the approximate eigenpairs decreases significantly as the accuracy tolerance becomes larger.

7 Future work

This dissertation addresses many important issues in the implementation of a parallel approximate eigensolver for real symmetric matrices, based on the PBD&C algorithm. Further improvements are possible. We recognize a few promising frontiers.

1) Adaptive eigensolver.

   Based on matrix structure and accuracy requirement, the approximate eigensolver chooses eigen-decomposition and pre-processing algorithms correspondingly. The approximate eigensolver can be further developed into an adaptive eigensolver that detects matrix structure automatically and then chooses proper algorithms. The determination of matrix structure is a heuristic process and may involve redundant computation. Plenty of test matrices from real applications, as well as complexity analyses, are necessary to verify and adjust the adaptivity of eigensolver.

2) Fine-tuning of workload balance for parallel BD&C implementation.

   In the parallel block tridiagonal divide-and-conquer subroutine PDSBTDC, the position of the last merging operation is determined by both the computational complexity and workload balance. We will further investigate the possibility and benefit of applying this strategy to merging levels preceding the final merging operation.

3) Complete data parallel implementation of BD&C.

   The parallel implementation of the BD&C algorithm in this dissertation uses a mixed data/task parallelism. Processors are assigned to matrix sub-blocks according to their sizes. At each level of the parallel merging tree, subproblems are merged simultaneously. When each subproblem on the same level of the merging tree has approximately the same deflation rate, we would expect all processors to finish one level of the merging tree at the same time. However, we lose workload balance when deflation rate varies drastically on the same level of the merging tree. In addition, when the number of processors is very small in comparison to the

number of diagonal blocks, the position of the final merging can no longer be optimally determined by computational complexity and workload balance. For example, suppose we have only two processors, then the off-diagonal block for the final merging operation is the one at the middle of the matrix even if the off-diagonal block in the middle has a high rank.

One possible solution is a complete data parallel implementation. The first attempt to parallelize the BD&C algorithm used data parallelism [25]. Due to the storage scheme of the diagonal blocks and off-diagonal blocks, that implementation was not able to exploit high performance of optimized parallel matrix multiplication. From our experience, the penalty of not being able to use optimized level 3 BLAS subroutine may degrade performance more severely than moderate overhead of extra data communication. A new completely data parallel implementation of BD&C will also involve matrix redistribution so that all the efficient algorithms in the sequential BD&C can be directly applied to the parallel implementation.

Bibliography

Bibliography

[1] E. Anderson, Z. Bai, C. Bishof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney and D. Sorensen, *LAPACK User's Guide*, 3rd Edition, Society for Industrial and Applied Mathematics, 1999.

[2] E. Anderson, J. Dongarra and S. Ostrouchov, *Implementation Guide for LAPACK*, Technical Report UT-CS-91-138, University of Tennessee, Knoxville, TN, 1991.

[3] P. Arbenz, K. Gates and C. Sprenger, *A Parallel Implementation of the Symmetric Tridiagonal QR Algorithm*, Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computations, IEEE Computer Society Press, 1992, McLean, VA (1992), pp. 382--388.

[4] Y. Bai, W. N. Gansterer and R. C. Ward, *Block-Tridiagonalization of "Effectively" Sparse Symmetric Matrices*, ACM Trans. Math. Softw., 30 (2004), pp. 326 -- 352.

[5] Z. Bai, J. Demmel, J. Dongarra, A. Petitet, H. Robinson and K. Stanley, *The Spectral Decomposition of Nonsymmetric Matrices on Distributed Memory Computers*, SIAM J. Sci. Comput., 18 (1997), pp. 1446-1461.

[6] H. J. Bernstein and M. Goldstein, *Parallel Implementation of Bisection for the Calculation of Eigenvalues of a Tridiagonal Symmetric Matrices*, Computing, 37 (1986), pp. 85 -- 91.

[7] P. Bientinesi, I. S. Dhillon and R. A. van de Geijn, *A parallel Eigensolver for Dense Symmetric Matrices based on Multiple Relatively Robust Representations*, SIAM J. Sci. Comput., 27 (2005), pp. 43 -- 66.

[8] C. Bishof, B. Lang and X. Sun, *The SBR Toolbox -- Software for Successive Band Reduction*, ACM Trans. Math. Softw., 26 (2000), pp. 602-616.

[9] C. Bishof, M. Marques and X. Sun, *Parallel Bandreduction and Tridiagonalization*, Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, SIAM Press, Philadelphia, PA (1993), pp. 383 -- 390.

[10] C. Bishof and C. F. Van Loan, *The WY Representation for Products of Householder Matrices*, SIAM J. Sci. Comput., 8 (1987), pp. 2 -- 13.

[11] C. H. Bishof, B. Lang and X. Sun, *A Framework for Symmetric Band Reduction*, ACM Trans. Math. Softw., 26 (2000), pp. 581 -- 601.

[12] C. H. Bishof, B. Lang and X. Sun, *A Framework for Symmetric Band Reduction and Tridiagonalization*, Technical Report ANL/MCS-P586-0496, Argonne National Laboratory, Argonne, IL, 1996.

[13] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongorra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker and R. C. Whaley, *ScaLAPACK User's Guide*, SIAM, Philadelphia, PA, 1997.

[14] S. Browne, J. Dongarra, N. Garner, G. Ho and P. Mucci, *Portable Programming Interface for Performance Evaluation on Modern Processors*, The International Journal of High Performance Computing Applications, 14 (2000), pp. 189-204.

[15] S. D. Browne, C., G. Ho and P. Mucci, *PAPI: A Portable Interface to Hardware Performance Counters*, Proceedings of Department of Defense HPCMP Users Group Conference, 1999.

[16] J. Bunch, C. Nielsen and D. Sorensen, *Rank-one Modification of the Symmetric Eigenproblem*, Numer. Math., 31 (1978), pp. 31 -- 48.

[17] J. Callaway, *Quantum Theory of the Solid State*, Academic Press, Boston, 1991.

[18] S. Chakrabarti, J. Demmel and D. Yelick, *Modeling the Benefits of Mixed Data and Task Parallelism*, Proceeding of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (1995), pp. 74 -- 83.

[19] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker and R. C. Whaley, *A Proposal for a Set of Parallel Basic Linear Algebra Subprograms*, Technical Report CS-95-292, University of Tennessee, Knoxville, TN, 1995.

[20] J. Choi, J. Dongarra, R. Pozo and D. Walker, *ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers*, Proceedings of Fourth Symposium on the Frontiers of Massively Parallel Computation, McLean,

Virginia), IEEE Computer Society Press, Los Alamitos, California (1992), pp. 120 -- 127.

[21] J. Choi, J. Dongarra and D. Walker, *The Design of Parallel Dense Linear Algebra Software Library: Reduction to Hessenberg, Tridiagonal, and Bidiagonal Form*, Numerical Algorithms, 10, Nos. 3 & 4 (1995), pp. 379 -- 400.

[22] M. Chu, *A Simple Application of the Homotopy Method to Symmetric Eigenvalue Problems*, Linear Algebra and Appl., 59 (1984), pp. 85 -- 90.

[23] H. L. Crane, Jr., N. E. Gibbs, W. G. Poole, Jr. and P. K. Stockmeyer, *Matrix Bandwidth and Profile Reduction*, ACM Trans. Math. Softw., 2 (1976), pp. 375 -- 377.

[24] J. J. M. Cuppen, *A Divide and Conquer Method for the Symmetric Tridiagonal Eigenproblem*, Numer. Math., 36 (1981), pp. 177 -- 195.

[25] R. M. Day, *A Coarse-Grain Parallel Implementation of the Block-Tridiagonal Divide-and-Conquer Algorithm for Symmetric Eigenproblems*, Master Thesis, University of Tennessee, Knoxville, TN, 2003.

[26] J. Demmel, *Applied Numerical Linear Algebra*, SIAM, Philadelphia, PA, 1997.

[27] J. Demmel and A. McKenney, *A Test Matrix Generation Suite*, Courant Institute of Mathematical Sciences, New York, 1989.

[28] M. J. S. Dewar and W. Thiel, *Ground States of Molecules, 38. The MNDO Method. Approximations and Parameters*, J. Amer. Chem. Soc., 99 (1977), pp. 4899-4907.

[29] I. S. Dhillon, *A new (N^2) Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*, PhD. Thesis, University of California, Berkeley, CA, 1997.

[30] I. S. Dhillon and B. N. Parlett, *Multiple Representation to Compute Orthogonal Eigenvectors of Symmetric tridiagonal Matrices*, Linear Algebra and Appl., 387 (2004), pp. 1 -- 28.

[31] I. S. Dhillon and B. N. Parlett, *Orthogonal Eigenvectors and Relative Gaps*, SIAM J. Matrix Anal. Appl., 25 (2004), pp. 858 -- 899.

[32] I. S. Dhillon, B. N. Parlett and C. Vömel, *LAPACK Working Note: the Design and Implementation of the MRRR Algorithm*, Technical Report UCB/CSD-04-1346, Computer Science Division, University of California at Berkeley, Berkeley, CA, 2004.

[33] J. Dongarra, J. Du Croz, I. S. Duff and S. Hammarling, *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Softw., 16 (1990), pp. 1 -- 17, 18 -- 28.

[34] J. Dongarra, J. Du Croz and S. Hammarling, *An Extended Set of Basic Linear Algebra Subprogram*, ACM Trans. Math. Softw., 14 (1988), pp. 18 -- 32.

[35] J. Dongarra, S. Hammarling and D. Sorensen, *Block Reduction of Matrices to Condensed Forms for Eigenvalue Computation*, J. Comp. Appl. Math., 27 (1989), pp. 215 -- 227.

[36] J. Dongarra and D. Sorensen, *A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. 139 -- 154.

[37] J. Dongarra and R. van de Geijn, *Reduction to Condensed Form for the Eigenvalue Problem on Distributed Memory Architectures*, Parallel Computing, 18 (1992), pp. 973 -- 982.

[38] T. H. Dunigan, Jr., *ORNL IBM Power4 (p690) Evaluation*, www.cs.ornl.gov/~dunigan/sp4.

[39] K. V. Fernando and B. N. Parlett, *Accurate Singular Values and Differential qd Algorithms*, Numer. Math., 67 (1994), pp. 191 -- 229.

[40] G. J. F. Francis, *The QR Transformation: A Unitary Analogue to the LR Transformation, Parts I and II*, Comput. J., 4 (1961), pp. 265 -- 271, 332 -- 345.

[41] W. N. Gansterer, D. F. Kvasnicka and C. W. Ueberhuber, *Multi-sweep Algorithms for the Symmetric Eigenproblem*, Lecture Notes in Computer Science, Vol. 1573 (1998), pp. 20 -- 28.

[42] W. N. Gansterer, R. C. Ward and R. P. Muller, *An Extension of the Divide-and-Conquer Method for a Class of Symmetric Block-tridiagonal Eigenproblems*, ACM Trans. Math. Softw., 28 (2002), pp. 45 -- 58.

[43] W. N. Gansterer, R. C. Ward, R. P. Muller and W. A. Goddard, III, *Computing Approximate Eigenpairs of Symmetric Block Tridiagonal Matrices*, SIAM J. Sci. Comput., 25 (2003), pp. 65 -- 85.

[44] K. Gates and P. Arbenz, *Parallel Divide and Conquer Algorithms for the Symmetric Tridiagonal Eigenproblem*, Technical Report, Institute for Scientific Computing, ETH Zurich, 1994.

[45] N. E. Gibbs, W. G. Poole, Jr. and P. K. Stockmeyer, *A Comparison of Several Bandwidth and Profile Reduction Algorithms*, ACM Trans. Math. Softw., 2 (1976), pp. 322 -- 330.

[46] N. E. Gibbs, W. G. J. Poole and P. K. Stockmeyer, *An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix*, SIAM J. Numer. Anal., 13 (1976), pp. 236 -- 250.

[47] W. J. Givens, *Numerical Computation of the Characteristic Values of a Real Symmetric Matrix*, Technical Report ORNL-1574, Oak Ridge National Lab, 1954.

[48] G. H. Golub, *Some Modified Matrix Eigenvalue Problems*, SIAM Review, 15(2) (1973), pp. 318 -- 334.

[49] G. H. Golub and C. F. van Loan, *Matrix Computations*, John Hopkins University Press, Baltimore /London, 1996.

[50] M. Gu, *Studies in Numerical Linear Algebra*, PhD. Thesis, Yale University, New Haven, Connecticut, 1993.

[51] M. Gu and S. C. Eisenstat, *A Divide-and-Conquer Algorithm for the Symmetric Tridiagonal Eigenproblem*, SIAM J. Matrix Anal. Appl., 16 (1995), pp. 172 -- 191.

[52] M. Gu and S. C. Eisenstat, *A Stable and Efficient Algorithm for the Rank-one Modification of the Symmetric Eigenproblem*, SIAM J. Matrix Anal. Appl., 15(4) (1994), pp. 1266 -- 1276.

[53] M. T. Heath, E. NG and B. W. Peyton, *Parallel Algorithms for Sparse Linear Systems*, Parallel Algorithms for Matrix Computations, SIAM Press, philadelphia, PA (1990), pp. 83 -- 124.

[54] S. Huss-Lederman, A. Tsao and T. Turnbull, *A Parallelizable Eigensolver for Real Diagonalizable Matrices with Real Eigenvalues*, SIAM J. Sci. Comput., 18 (1997), pp. 869 -- 885.

[55] S. Huss-Lederman, A. Tsao and G. Zhang, *A Parallel Implementation of the Invariant Subspace decomposition Algorithm for Dense Symmetric Matrices*, Proceeding of the Sixth SIAM conference on Parallel Processing for Scientific Computing in Norfork, VA, 1993.

[56] IBM, *Parallel ESSL for AIX, Guide and Reference, V2.3*, 2003.

[57] I. C. F. Ipesn, *Computing an Eigenvector with Inverse Iteration*, SIAM Review, 39 (1997), pp. 254 -- 291.

[58] I. C. F. Ipesn and E. R. Jessup, *Solving the Symmetric Tridiagonal EIgenvalue Problem on the Hypercube*, SIAM J. Sci. Stat. Comput., 11(2) (1990), pp. 203 -- 229.

[59] E. R. Jessup and I. C. F. Ipesn, *Improving the Accuracy of Inverse Iteration*, SIAM J. Sci. Stat. Comput., 13 (1992), pp. 550 -- 572.

[60] G. Karypis and V. Kumar, *A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering*, J. Parallel Distrib. Comput., 48 (1998), pp. 71 -- 95.

[61] L. Kaufman, *A Parallel QR Algorithm for the Symmetric Tridiagonal Eigenvalue Problem*, J. Parallel Distrib. Comput., 23 (1994), pp. 429 -- 434.

[62] W. Kohn and L. J. Sham, *Self-Consistent Equations Including Exchange and Correlation Effects*, Phys. Rev. A, 140 (1965), pp. A1133–A1138.

[63] D. Kuck and A. Sameh, *A Parallel QR Algorithm for Symmetric Tridiagonal Matrices*, IEEE Trans. Computers, C-26, 1977.

[64] J. G. Lewis, *The Gibbs-Poole-Stockmeyer and Gibbs-King Algorithms for Reordering Sparse Matrices*, ACM Trans. Math. Softw., 8 (1982), pp. 190 -- 194.

[65] K. Li and T. Y. Li, *An Algorithm for Symmetric Tridiagonal Eigenproblems -- Divide and Conquer with Homotopy Continuation*, SIAM J. Sci. Comput., 14 (1993), pp. 735 -- 751.

[66] R.-C. Li, *Solving the Secular Equation Stably and Efficiently*, Technical Report, Department of Mathematics, University of California, Berkeley, CA, 1993.

[67] T. Y. Li and N. Rhee, *Homotopy Algorithm for Symmetric Eigenvalue Problems*, Numer. Math., 55 (1989), pp. 265 -- 280.

[68] T. Y. Li, H. Zhang and X. Sun, *Paralle Homotopy Algorithm for the Symmetric Tridiagonal Eigenvalue Problem*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 469 -- 487.

[69] S.-S. Lo, B. Phillipe and A. Sameh, *A Multiprocessor Algorithm for the Symmetric Eigenproblem*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. 155 -- 165.

[70] O. Madelung, *Introduction to Solid-State Theory*, 2nd Ed., Springer-Verlag, Berlin, 1996.

[71] N. H. March, W. H. Young and S. Sampanthar, *The Many-body Problem in Quantum Mechanics*, Dover Publications, Mineola, NY, 1995.

[72] A. Melman, *A numerical Comparison of Methods for Solving Secular Equations*, J. Comp. Appl. Math., 86(1) (1997), pp. 237 -- 249.

[73] M. Oettli, *The Homotopy Method Applied to the Symmetric Eigenproblem*, PhD. Dissertation, ETH Zurith, 1996.

[74] M. Oettli, *A Robust, Parallel Homotopy Algorithm for the Symmetric Tridiagonal Eigenproblem*, SIAM J. Sci. Comput., 20 (1999), pp. 1016 -- 1032.

[75] B. N. Parlett, *Invariant Subspaces for Tightly Clustered Eigenvalues of Tridiagonals*, BIT, 36 (1996), pp. 542 -- 562.

[76] B. N. Parlett, *Spectral Sensitivity of Products of Bidiagonals*, Linear Algebra and Appl., 275 -- 276 (1998), pp. 417 --431.

[77] B. N. Parlett, *The Symmetric Eigenvalue Problem*, SIAM Press, Philadelphia, PA, 1997.

[78] R. G. Parr and W. Yang, *Density-Functional Theory of Atoms and Molecules*, Oxford University Press, 1994.

[79] J. P. Perdew, K. Burke and M. Ernzerhof, *Generalized Gradient Approximation Made Simple*, Phys. Rev. Lett., 77 (1996), pp. 3865 -- 3868.

[80] G. Peters and J. H. Wilkinson, *The Calculation of Specified Eigenvectors by Inverse Iteration*, Contribution II/18, Volumn II of Handbook of Automatic Computation, Springer-Verlag, New York/Heidelberg/Berlin, 1971.

[81] J. A. Pople and D. L. Beveridge, *Approximate Molecular Orbital Theory*, 1st ed., McGraw-Hill, New York, 1970.

[82] J. A. Pople and G. A. Segal, *Approximate Self-Consistent Molecular Orbital Theory. II. Calculations with Complete Neglect of Differential Overlap*, 43 (1965), pp. 136.

[83] J. A. Pople and G. A. Segal, *Approximate Self-Consistent Molecular Orbital Theory. III. CNDO Results for AB2 and AB3 Systems*, J. Chem. Physics, 44 (1966), pp. 3289.

[84] C. H. Reinsh, *A Stable Rational QR Algorithm for the Computation of the Eigenvalues of a Hermitian, Tridiagonal Matrix*, Numer. Math., 25 (1971), pp. 591 -- 597.

[85] H. Rutishauser, *Der Quotienten-Differenzen-Algorithmus*, Z. Angew. Math. Phys., 5 (1954), pp. 223--152.

[86] H. Rutishauser, *Lectures on Numerical Mathematics*, Birkhäuser, Boston, 1990.

[87] J. Rutter, *A Serial Implementation of Cuppen's Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem*, Technical Report CS-94-225, Department of Computer Science, University of Tennessee, Knoxville, TN, 1994.

[88] R. Schreiber and C. F. Van Loan, *A Storage-Efficient WY Representation for Products of Householder Transformations*, SIAM J. Sci. Comput., 10 (1989), pp. 53 -- 57.

[89] D. Sorensen and P. T. P. Tang, *On the Orthogonality of Eigenvectors Computed by Divide and Conquer Methods Techniques*, SIAM J. Numer. Anal., 28 (1991), pp. 1752 -- 1775.

[90] W. P. Su, J. R. Schrieffer and A. J. Heeger, *Soliton Excitations in Polyacetylene*, Phys. Rev. B, 22 (1980), pp. 2099 -- 2111.

[91] A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry*, Dover Publications, Mineola, NY, 1996.

[92] F. Tisseur and J. Dongarra, *Parallelizing the Divide and Conquer Algorithm for the Symmetric Tridiagonal Eigenvalue Problem on Distributed Memory Architectures*, SIAM J. Sci. Comput., 20 (1999), pp. 2223 -- 2236.

[93] R. van de Geijn, *Deferred Shifting Schemes for Parallel QR Methods*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 180 -- 194.

[94] R. van de Geijn and J. Watts, *SUMMA: Scalable Universal Matrix Multiplication Algorithm*, Concurrency: Practice and Experience, 9 (1997), pp. 255 -- 274.

[95] R. C. Ward, Y. Bai and J. Pratt, *Performance of Parallel Eigensolvers on Electronic Structure Calculations*, Technical Report, UT-CS-05-560, University of Tennessee, Knoxville, TN, 2005.

[96] J. H. Wilkinson, *The Algebraic EIgenvalue Problem*, Oxford University Press, Oxford, 1965.

[97] J. H. Wilkinson, *Global Convergence of the Tridiagonal QR Algorithm with Origin Shifts*, Linear Algebra and Appl., 1 (1968), pp. 409 -- 420.

[98] P. H. Worley, T. H. Dunigan, Jr., M. R. Fahey, J. B. White, III and A. S. Bland, *Early Evaluation of the IBM p690*, Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, Baltimore, Maryland (2002), pp. 1 -- 21.

[99] Y. J. Wu, P. A. Alpatov, C. H. Bishof and R. A. van de Geijn, *A Parallel Implementation of Symmetric Band Reduction Using PLAPACK*, Proceedings of Scalable Parallel Library Conference, Starkville, Mississippi, 1996.

Appendix

Appendix Complete numerical test results

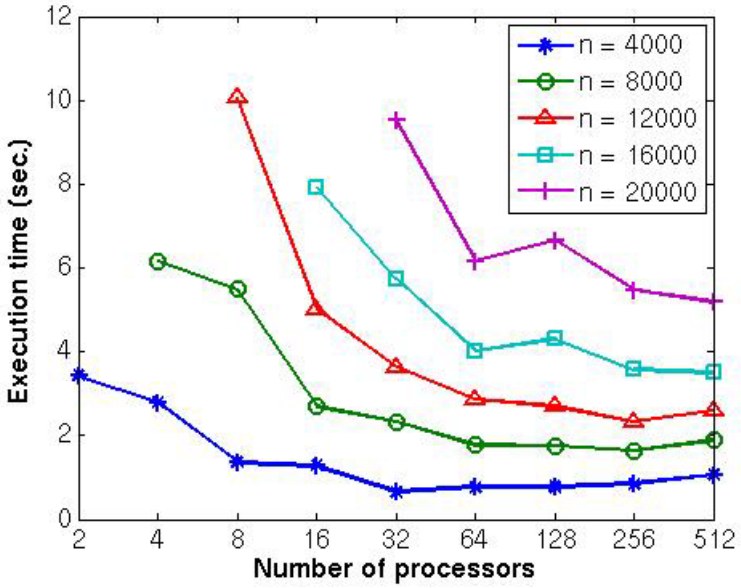

Figure A. 1 Execution of PDSBTDC using **P-clu0** matrices, $\tau = 10^{-6}$.

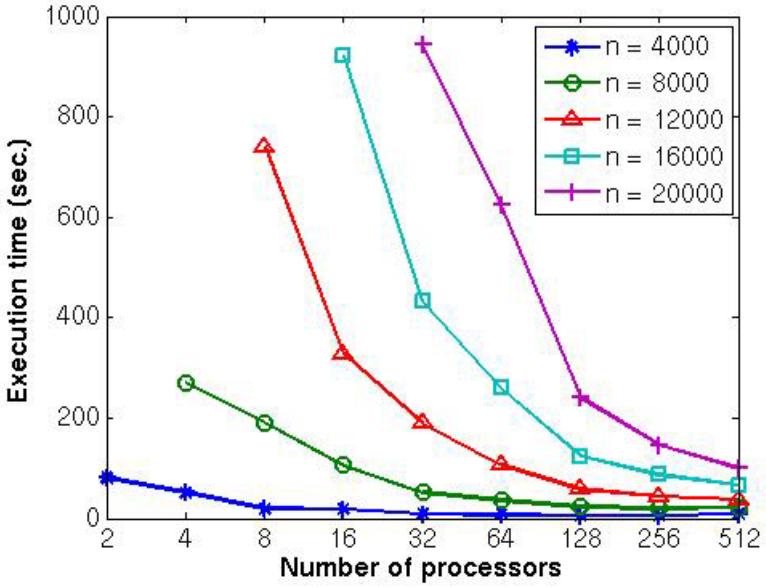

Figure A. 2 Execution of PDSYEVD using **P-clu0** matrices.

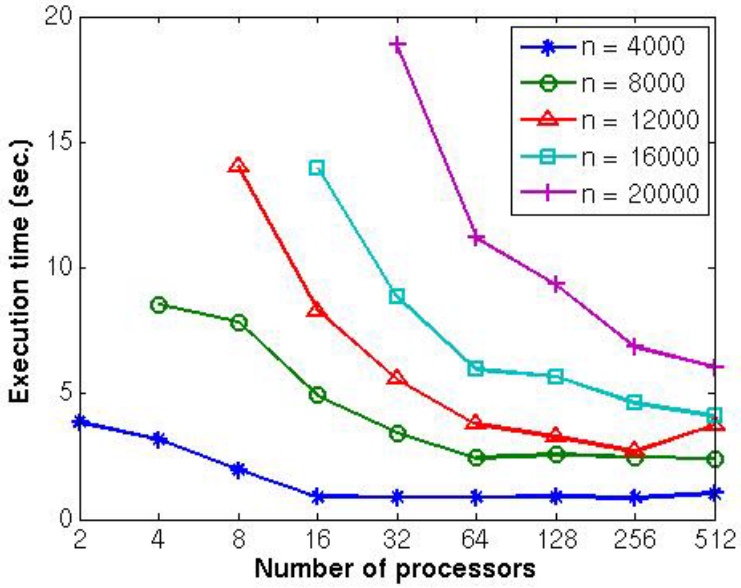

Figure A. 3 Execution of PDSBTDC using **P-clu1** matrices, $\tau = 10^{-6}$.

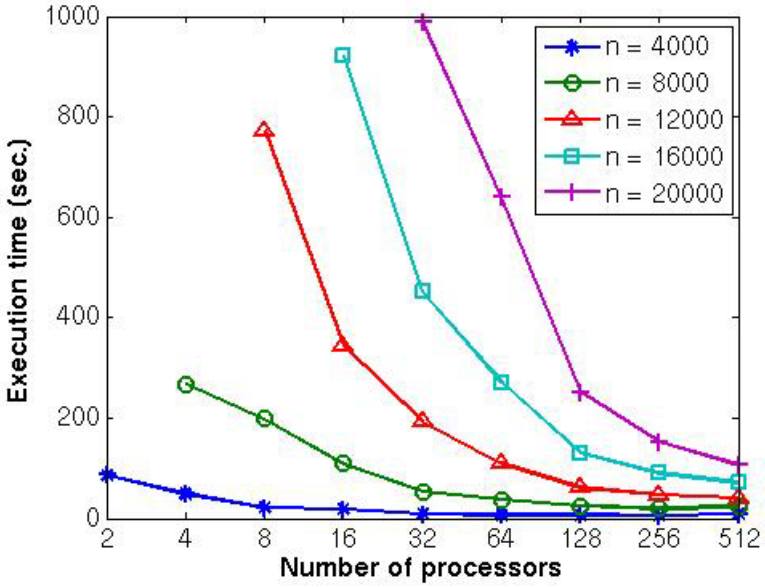

Figure A. 4 Execution of PDSYEVD using **P-clu1** matrices.

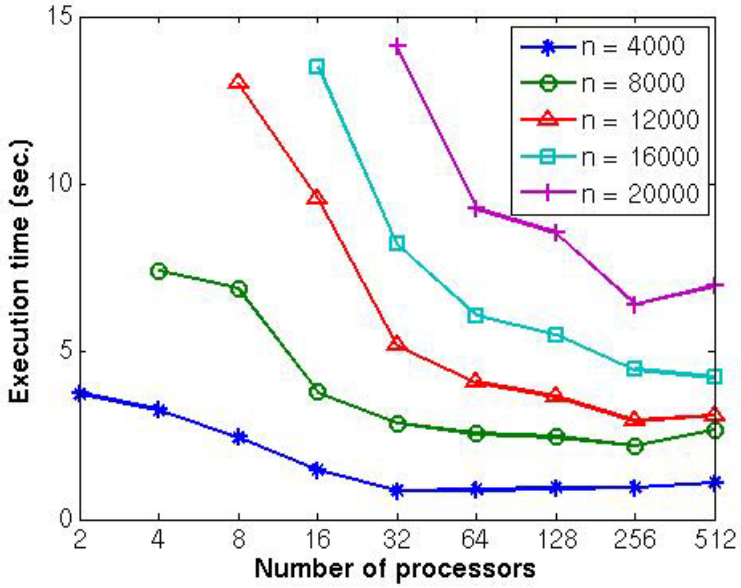

Figure A. 5 Execution of PDSBTDC using **P-geom** matrices, $\tau = 10^{-6}$.

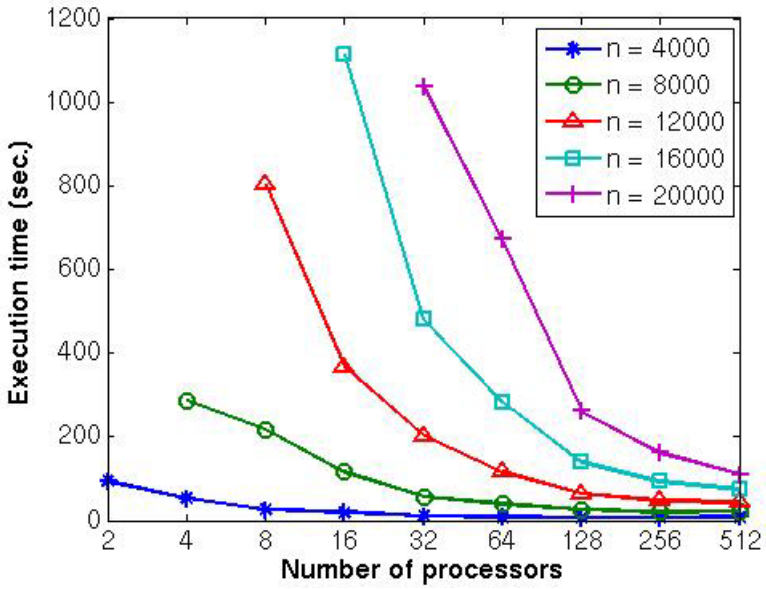

Figure A. 6 Execution of PDSYEVD using **P-geom** matrices.

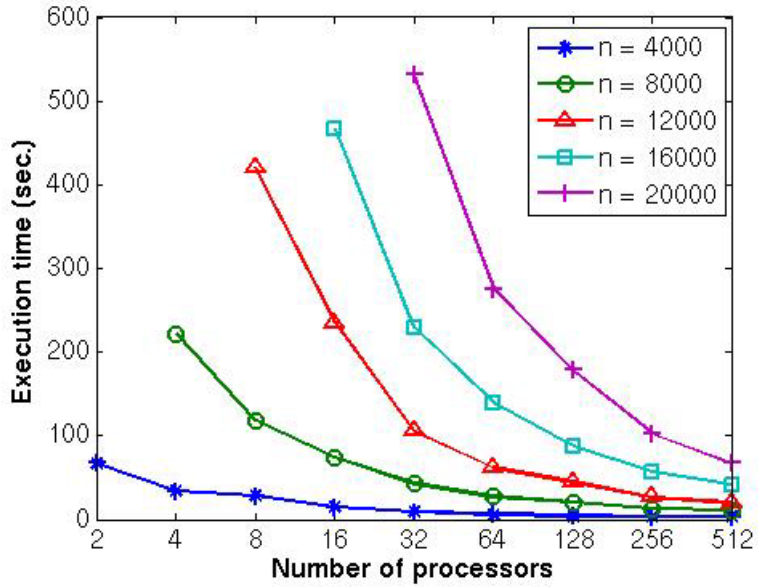

Figure A. 7 Execution of PDSBTDC using **P-arith** matrices, $\tau = 10^{-6}$.

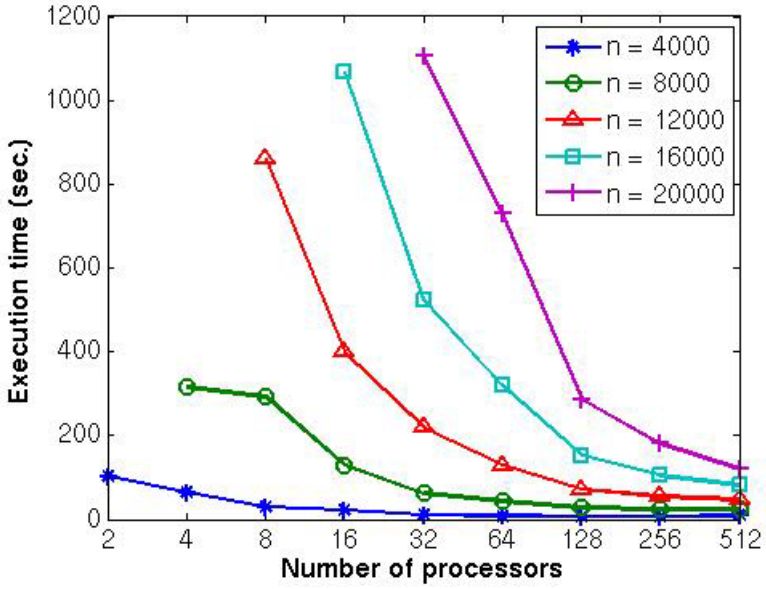

Figure A. 8 Execution of PDSYEVD using **P-arith** matrices.

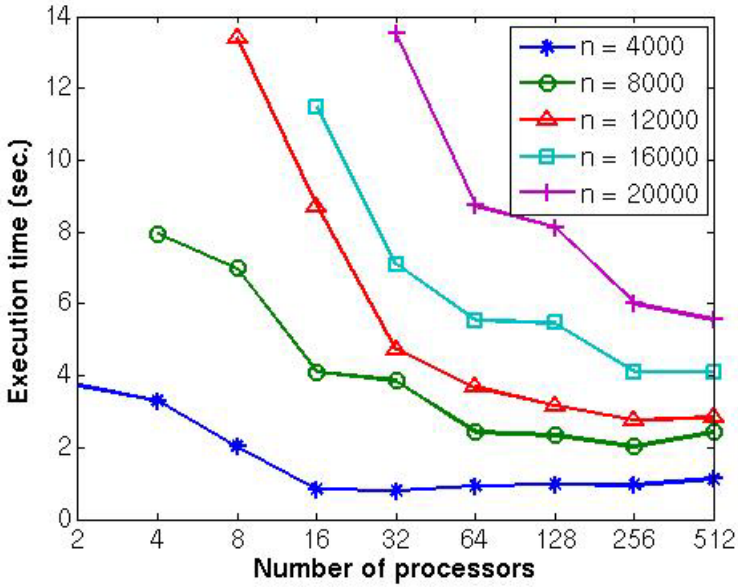

Figure A. 9 Execution of PDSBTDC using **P-log** matrices, $\tau = 10^{-6}$.

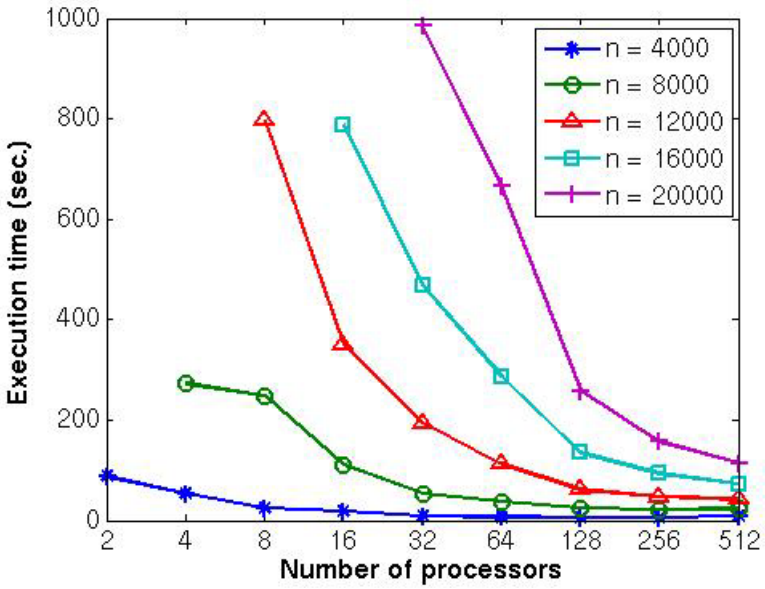

Figure A. 10 Execution of PDSYEVD using **P-log** matrices.

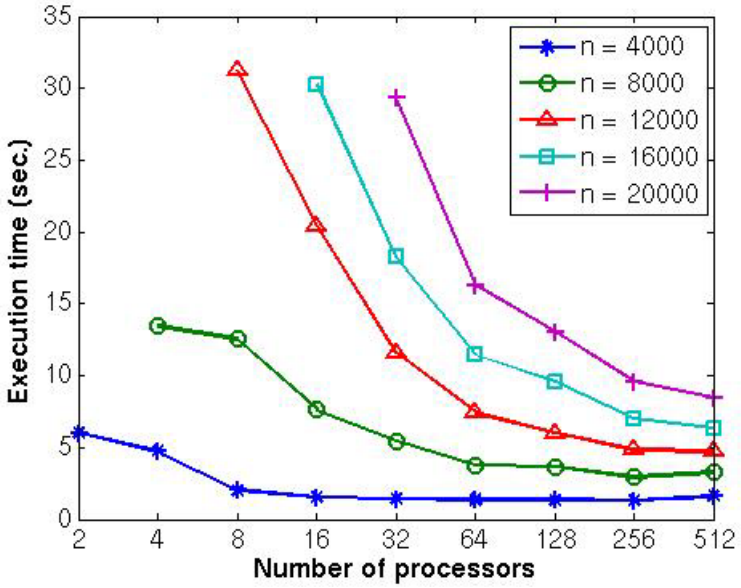

Figure A. 11 Execution of PDSBTDC using **P-rand** matrices, $\tau = 10^{-6}$.

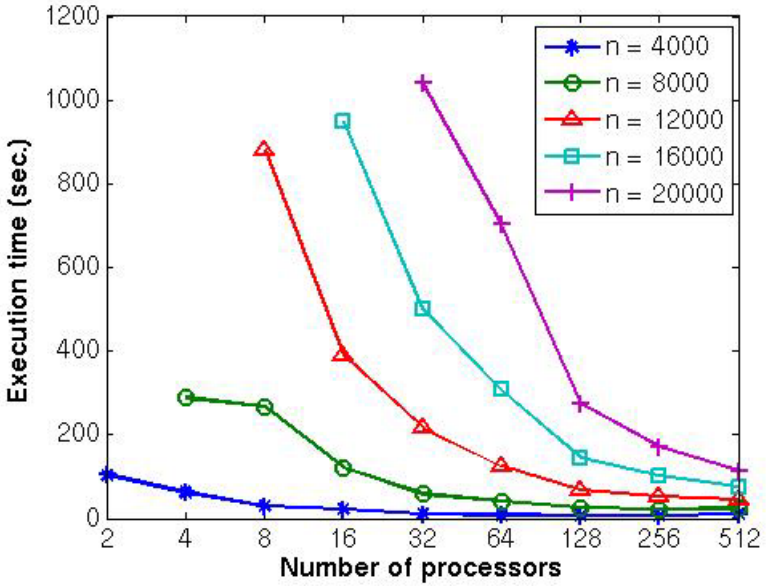

Figure A. 12 Execution of PDSYEVD using **P-rand** matrices.

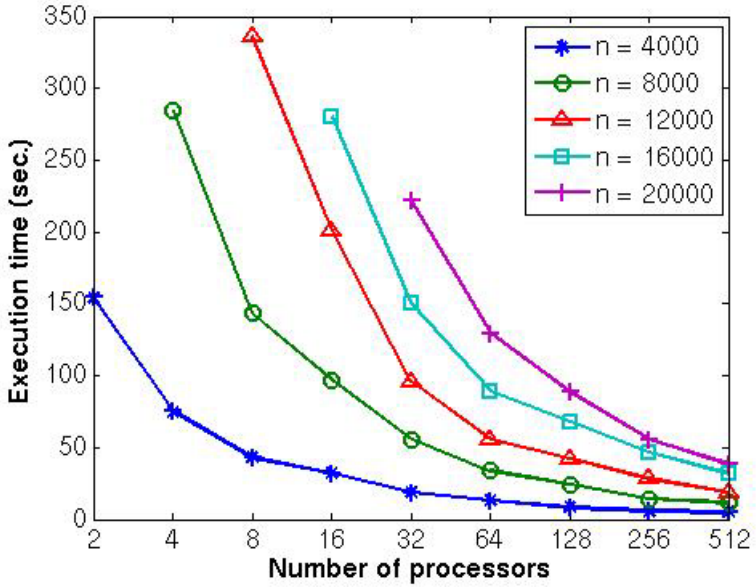

Figure A. 13 Execution of PDSBTDC using **R-bt** matrices, $\tau = 10^{-6}$.

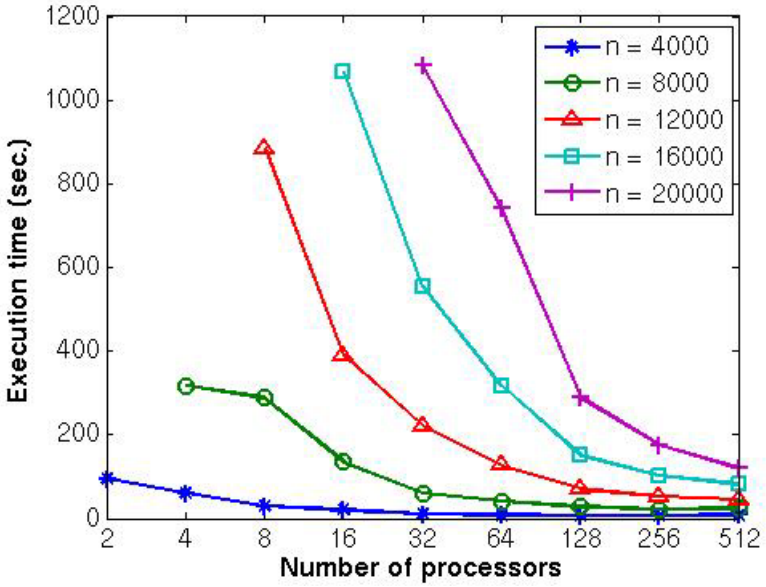

Figure A. 14 Execution of PDSYEVD using **R-bt** matrices.

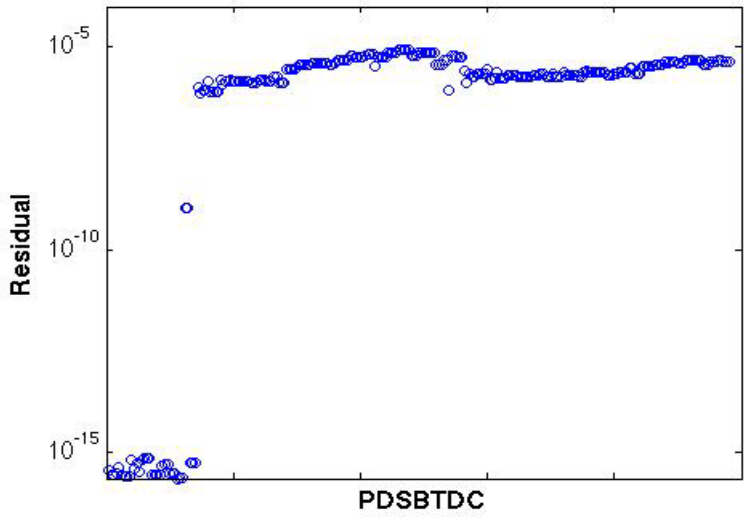

Figure A. 15 Scaled residual $\mathcal{R} = \max_{i=1,\cdots,n} \frac{\left\| A\hat{x}_i - \hat{\lambda}_i \hat{x}_i \right\|_2}{\left\| A \right\|_2}$ of PDSBTDC using **P-clu0**, **P-clu1**, **P-geom**, **P-arith**, **P-log**, **P-rand**, and **R-bt** matrices, $\tau = 10^{-6}$.

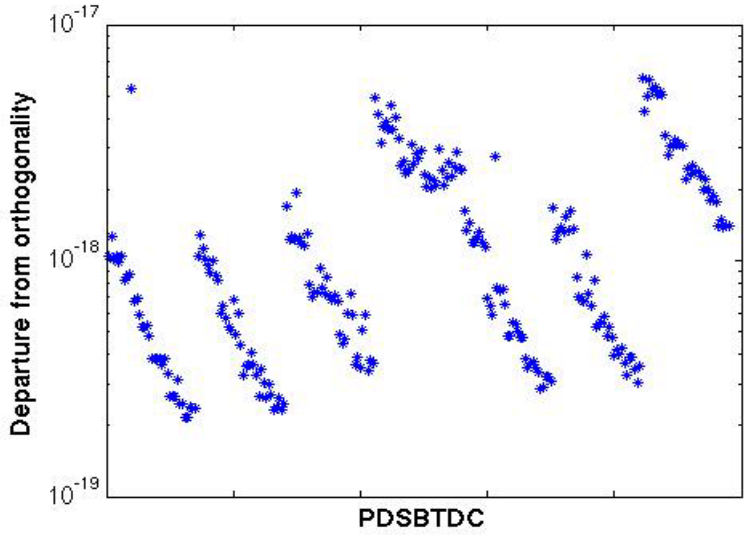

Figure A. 16 Departure from orthogonality $\mathcal{O} = \frac{\max_{i=1,\cdots,n} \left\| \left(\hat{X}^T \hat{X} - I \right) e_i \right\|_2}{n}$ of PDSBTDC using **P-clu0**, **P-clu1**, **P-geom**, **P-arith**, **P-log**, **P-rand**, and **R-bt** matrices, $\tau = 10^{-6}$.

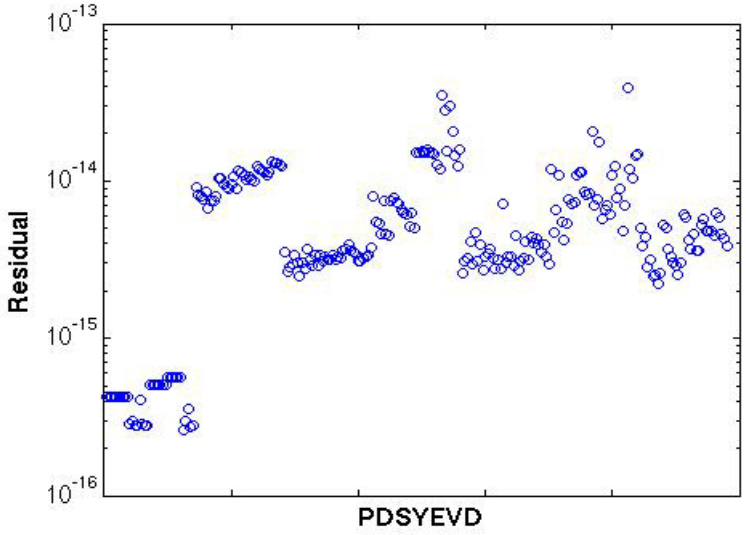

Figure A. 17 Scaled residual $\mathcal{R} = \max_{i=1,\cdots,n} \frac{\left\| A\hat{x}_i - \hat{\lambda}_i \hat{x}_i \right\|_2}{\left\| A \right\|_2}$ of PDSYEVD using **P-clu0**, **P-clu1**, **P-geom**, **P-arith**, **P-log**, **P-rand**, and **R-bt** matrices.

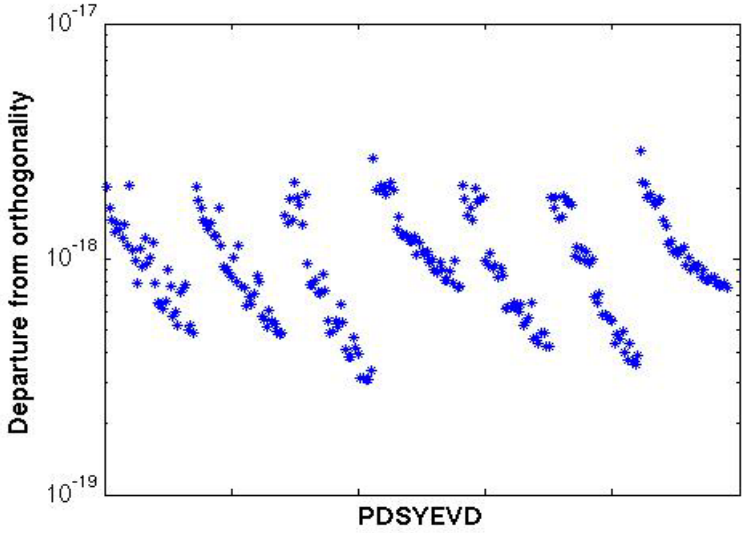

Figure A. 18 Departure from orthogonality $\mathcal{O} = \frac{\max_{i=1,\cdots,n} \left\| \left(\hat{X}^T \hat{X} - I \right) e_i \right\|_2}{n}$ of PDSYEVD using **P-clu0**, **P-clu1**, **P-geom**, **P-arith**, **P-log**, **P-rand**, and **R-bt** matrices.

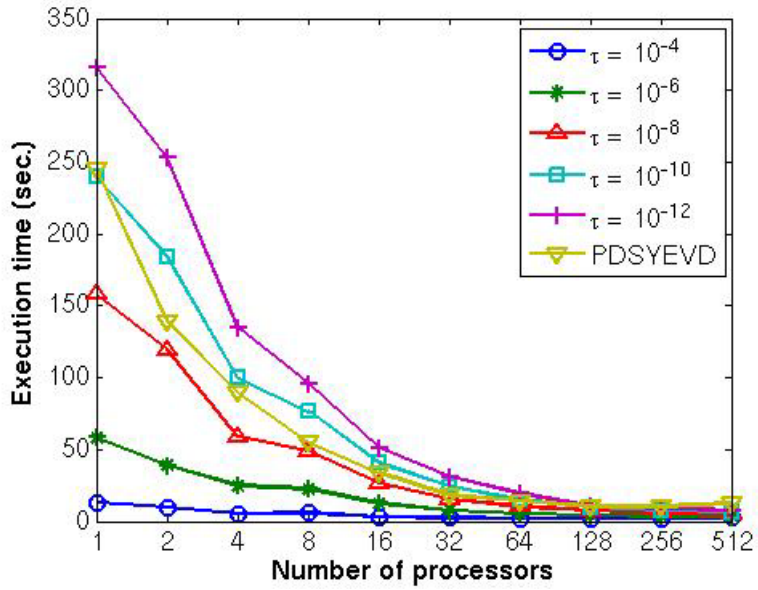

Figure A. 19 Execution time of PDSBTDC and PDSYEVD using matrix **A-ala**. Matrix size $n = 5,027$. Tolerance for PDSBTDC $\tau = 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}$.

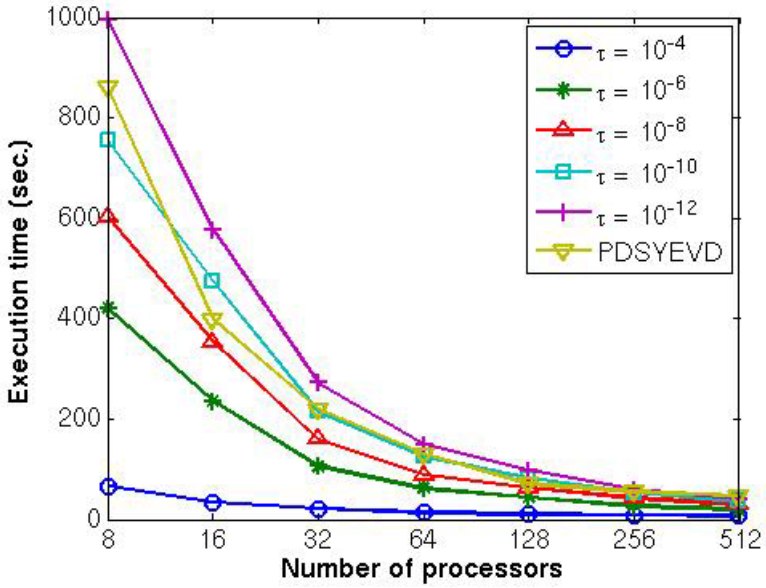

Figure A. 20 Execution time of PDSBTDC and PDSYEVD using **P-arith** matrix. Matrix size $n = 12,000$. Tolerance for PDSBTDC $\tau = 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}$.

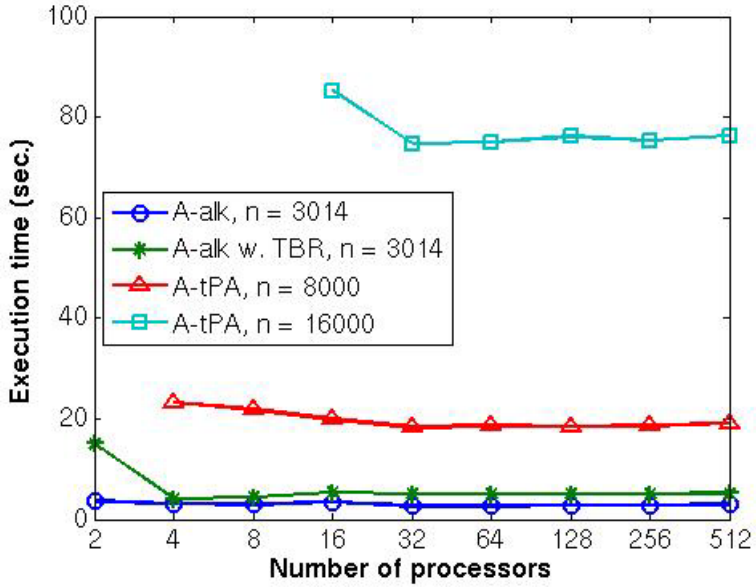

Figure A. 21 Execution time of PDSBTRI with $\tau = 10^{-4}$.

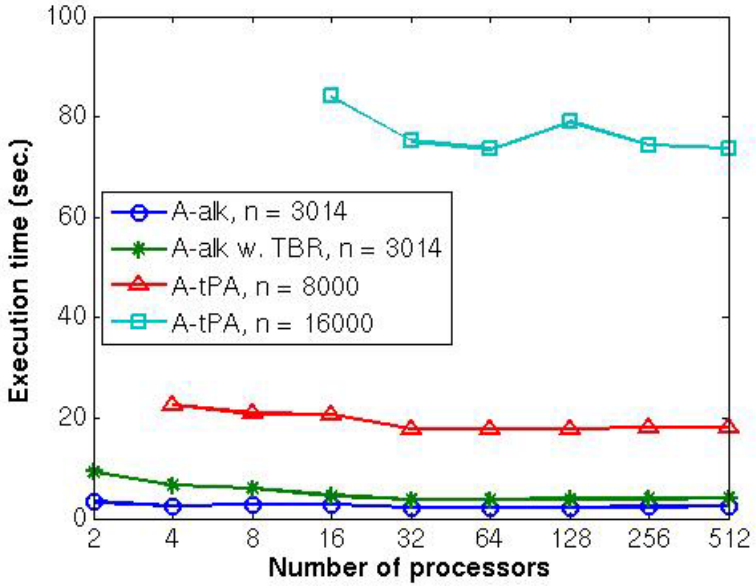

Figure A. 22 Execution time of PDSBTRI with $\tau = 10^{-8}$.

Table A.1 Scaled eigenvalue error $\left|\lambda(A)-\lambda(M)\right|/\|A\|$ of PDSBTRI.

Matrix	**Size**	**With TBR**	**Tolerance**	$\left\|\lambda(A)-\lambda(M)\right\|/\|A\|$
A-alk	3,014	No	10^{-4}	1.09×10^{-5}
		Yes	10^{-4}	5.79×10^{-6}
A-tPA	8,000	No	10^{-4}	1.09×10^{-5}
	16,000	No	10^{-4}	1.10×10^{-5}
A-alk	3,014	No	10^{-6}	3.55×10^{-7}
		Yes	10^{-6}	3.57×10^{-7}
A-tPA	8,000	No	10^{-6}	8.04×10^{-8}
	1,6000	No	10^{-6}	7.19×10^{-8}
A-alk	3,014	No	10^{-8}	3.42×10^{-9}
		Yes	10^{-8}	3.45×10^{-9}
A-tPA	8,000	No	10^{-8}	5.38×10^{-10}
	16,000	No	10^{-8}	5.26×10^{-10}

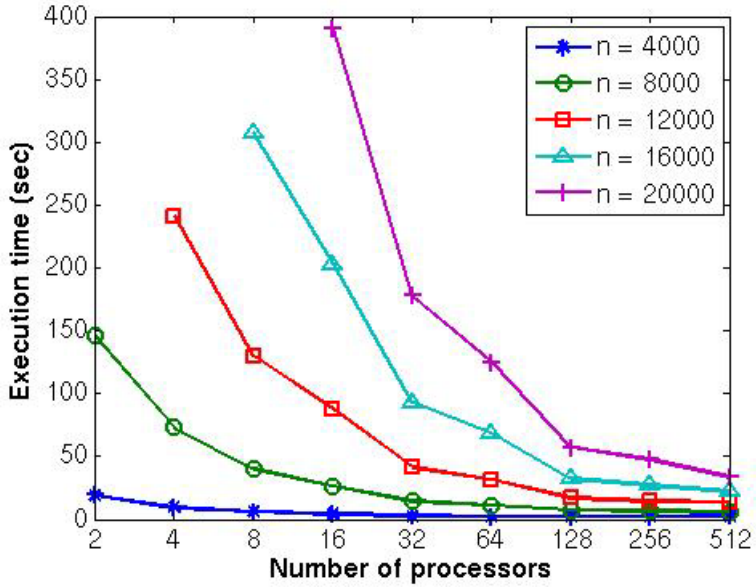

Figure A. 23 Execution time of PDSBTRD using **R-den** matrices.

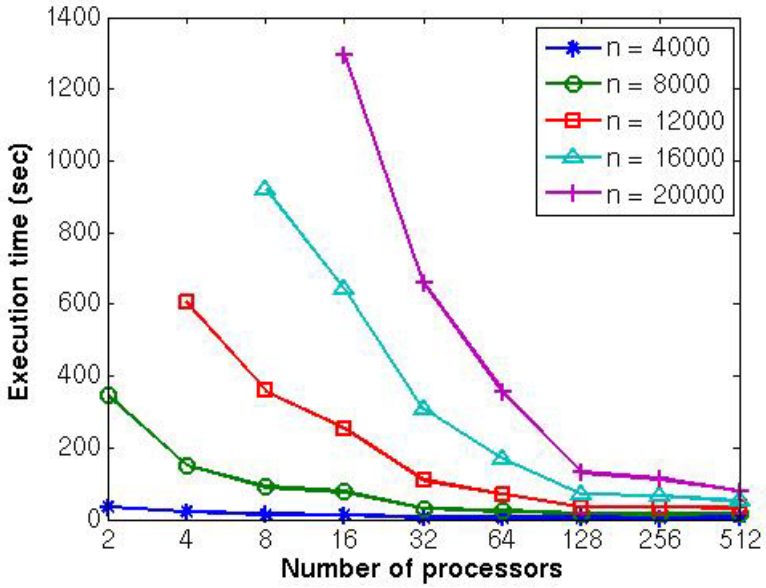

Figure A. 24 Execution time of PDSYTRD using **R-den** matrices.

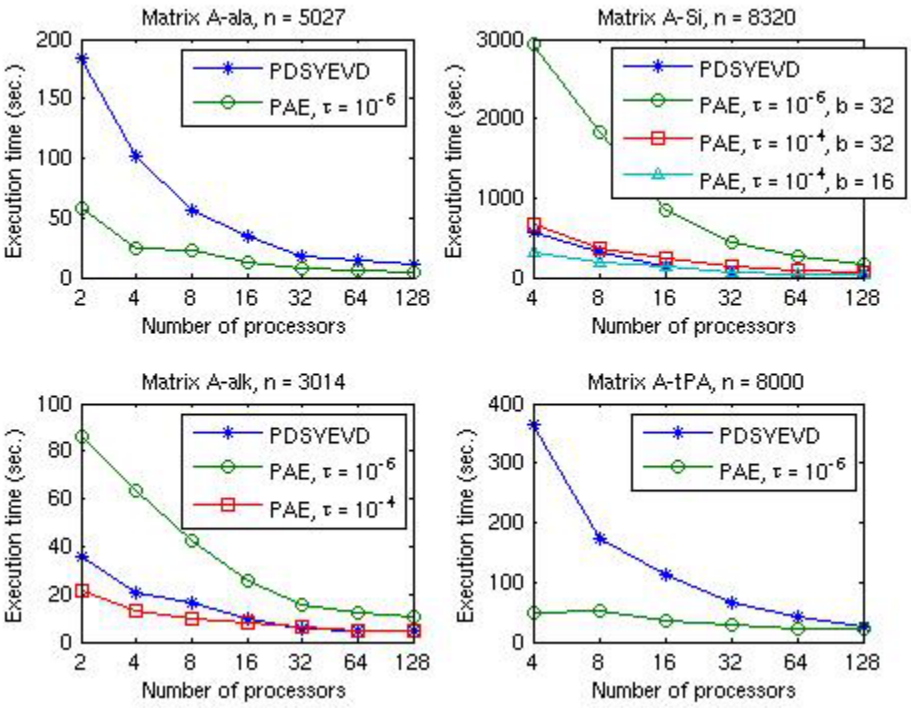

Figure A. 25 Execution time of parallel approximate eigensolver (PAE) and PDSYEVD.

Vita

Yihua Bai received a B.E. in Material Engineering from Jiao Tong University, Shanghai, China and a M.S. in Computer Science from University of Tennessee, Knoxville. She is currently pursuing her doctorate in Computer Science at the University of Tennessee, Knoxville.